AFRL-RI-RS-TR-2017-081

REQUIREMENTS PATTERNS FOR FORMAL CONTRACTS IN ARCHITECTURAL ANALYSIS AND DESIGN LANGUAGE (AADL) MODELS

ROCKWELL COLLINS

APRIL 2017

FINAL TECHNICAL REPORT

APPROVED FOR PUBLIC RELEASE; DISTRIBUTION UNLIMITED

STINFO COPY

AIR FORCE RESEARCH LABORATORY
INFORMATION DIRECTORATE

■ **AIR FORCE MATERIEL COMMAND** ■ **UNITED STATES AIR FORCE** ■ **ROME, NY 13441**

NOTICE AND SIGNATURE PAGE

Using Government drawings, specifications, or other data included in this document for any purpose other than Government procurement does not in any way obligate the U.S. Government. The fact that the Government formulated or supplied the drawings, specifications, or other data does not license the holder or any other person or corporation; or convey any rights or permission to manufacture, use, or sell any patented invention that may relate to them.

This report was cleared for public release by the 88th ABW, Wright-Patterson AFB Public Affairs Office and is available to the general public, including foreign nationals. Copies may be obtained from the Defense Technical Information Center (DTIC) (http://www.dtic.mil).

AFRL-RI-RS-TR-2017-081 HAS BEEN REVIEWED AND IS APPROVED FOR PUBLICATION IN ACCORDANCE WITH ASSIGNED DISTRIBUTION STATEMENT.

FOR THE CHIEF ENGINEER:

/ S /
STEVEN DRAGER
Work Unit Manager

/ S /
JOHN MATYJAS
Technical Advisor, Computing
& Communications Division
Information Directorate

This report is published in the interest of scientific and technical information exchange, and its publication does not constitute the Government's approval or disapproval of its ideas or findings.

REPORT DOCUMENTATION PAGE	*Form Approved* **OMB No. 0704-0188**

The public reporting burden for this collection of information is estimated to average 1 hour per response, including the time for reviewing instructions, searching existing data sources, gathering and maintaining the data needed, and completing and reviewing the collection of information. Send comments regarding this burden estimate or any other aspect of this collection of information, including suggestions for reducing this burden, to Department of Defense, Washington Headquarters Services, Directorate for Information Operations and Reports (0704-0188), 1215 Jefferson Davis Highway, Suite 1204, Arlington, VA 22202-4302. Respondents should be aware that notwithstanding any other provision of law, no person shall be subject to any penalty for failing to comply with a collection of information if it does not display a currently valid OMB control number.
PLEASE DO NOT RETURN YOUR FORM TO THE ABOVE ADDRESS.

1. REPORT DATE *(DD-MM-YYYY)*	**2. REPORT TYPE**	**3. DATES COVERED** *(From - To)*
APRIL 2017	FINAL TECHNICAL REPORT	OCT 2015 – OCT 2016

4. TITLE AND SUBTITLE REQUIREMENTS PATTERNS FOR FORMAL CONTRACTS IN ARCHITECTURAL ANALYSIS AND DESIGN LANGUAGE (AADL) MODELS	**5a. CONTRACT NUMBER** FA8750-16-C-0018
	5b. GRANT NUMBER N/A
	5c. PROGRAM ELEMENT NUMBER 63781D
6. AUTHOR(S) John Backes	**5d. PROJECT NUMBER** ASET
	5e. TASK NUMBER 15
	5f. WORK UNIT NUMBER RC
7. PERFORMING ORGANIZATION NAME(S) AND ADDRESS(ES) Rockwell Collins 7805 Telegraph Road Bloomington, MN 55438	**8. PERFORMING ORGANIZATION REPORT NUMBER**
9. SPONSORING/MONITORING AGENCY NAME(S) AND ADDRESS(ES) Air Force Research Laboratory/RITA 525 Brooks Road Rome NY 13441-4505	**10. SPONSOR/MONITOR'S ACRONYM(S)** AFRL/RI
	11. SPONSOR/MONITOR'S REPORT NUMBER AFRL-RI-RS-TR-2017-081

12. DISTRIBUTION AVAILABILITY STATEMENT

Approved for Public Release; Distribution Unlimited. PA# 88ABW-2017-1665
Date Cleared: 10 Apr 2017

13. SUPPLEMENTARY NOTES

14. ABSTRACT

English language requirements are often used to specify the behavior of complex cyber-physical systems. The process of transforming these requirements to a formal specification language is often challenging, especially if the specification language does not contain constructs analogous to those used in the original requirements. For example, requirements often contain real-time constraints, but many specification languages for model checkers have discrete time semantics. Work in specification patterns helps to bridge these gaps, allowing straightforward expression of common requirements patterns in formal languages. In this report we demonstrate how we support real-time specification patterns in the Assume Guarantee Reasoning Environment (AGREE) using observers. We demonstrate that there are subtle challenges, not mentioned in previous literature, to express real-time patterns accurately using observers. We demonstrate that these patterns are sufficient to model real-time requirements for a real-world avionics systems.

15. SUBJECT TERMS

Cyberphysical Systems, Formal Methods, Requirements Patterns, AADL, Assume Guarantee Reasoning Environment

16. SECURITY CLASSIFICATION OF:			**17. LIMITATION OF ABSTRACT**	**18. NUMBER OF PAGES**	**19a. NAME OF RESPONSIBLE PERSON** STEVEN DRAGER
a. REPORT U	**b. ABSTRACT** U	**c. THIS PAGE** U	UU	42	**19b. TELEPHONE NUMBER** *(Include area code)* N/A

Table of Contents

Table of Figures

1. Summary

Modern aircraft are complex cyber-physical systems with safety and security requirements that must be satisfied by their onboard software. As these systems have grown in complexity, their verification has become the single most costly development activity [1]. The verification costs of even more complex systems in the future will impact safety, not just through an increasing incidence of errors and unforeseen interactions, but by delaying and preventing the deployment of crucial safety functions.

Rockwell Collins has been addressing these challenges by developing compositional reasoning methods that permit the verification of systems that exceed the complexity limits of current approaches. Our approach is based on:

- Modeling the system architecture using standard notations that are usable by systems and software engineers.
- Developing a sophisticated translation framework that automates the translation of system architecture models for analysis by powerful general-purpose verification engines such as Satisfiability Modulo Theories (SMT)-based model checkers.
- Developing techniques for compositional verification based on the system architecture to divide the verification task into manageable, reusable pieces.

This approach has the potential to significantly reduce verification costs by identifying and correcting system design errors early in the life cycle rather than waiting until system integration. However, formally verifying requirements for complex cyber physical systems is a challenging task. Practitioners will often make simplifying assumptions about the system in order to make verification tractable.

A common simplification is to assume that all of the components of a system execute synchronously in order to decrease the verification effort. Yet, real systems often contain components that execute via different clock domains. Assumptions about synchrony are likely not sound, and they can prevent engineers from discovering serious vulnerabilities, errors, and/or race conditions. Skilled practitioners may be able to make sound abstractions about timing information, such as modeling the systems execution with quasi-synchronous constraints. In a previous effort, we developed a framework to model and reason about quasi-synchronous systems. We successfully demonstrated our approach on a number of models that were based on actual examples seen in industry. However, this approach contained a couple of pitfalls:

1. It can be difficult to determine how to formalize a natural language requirement in terms of quasi-synchronous constraints.
2. It is not feasible to compositionally analyze systems containing quasi-synchronous constraints.

To solve these issues we opted to alter the specification language of our compositional reasoning tool called the Assume Guarantee Reasoning Environment (AGREE)[1]. The new specification language allows users to write requirements using specification patterns. This reduces the burden of determining how to specify requirements, and it also increases the likelihood of requirements being formalized correctly. Furthermore, the patterns allow users to specify requirements with respect to real-time. This not only makes the meaning of a formalization more clear, it also allows requirements to compose more easily. For the remainder of this report we refer to this effort as the Contract Requirements Patterns (CRP) project.

[1] AGREE stands for Assume-Guarantee Reasoning Environment

2. Introduction

Specification patterns [2, 3] are an approach to ease the construction of formal specifications from natural language requirements. These patterns describe how common reasoning patterns in English language requirements can be represented in (sometimes complex) formulas in a variety of formalisms. Following the seminal work of Dwyer [2] for discrete time specification patterns, a variety of real-time specification pattern taxonomies have been developed [3–7]. An example of a timed specification pattern expressible in each is: "Globally, it is always the case that if P holds, then S holds between low and high time unit(s)." In most of this work, the specification patterns are mapped to real-time temporal logics, such as Timed Computational Tree Logic (TCTL) [8], Metric Temporal Logic (MTL) [9], Real-Time Graphical Interval Logic (RTGIL) [10], and Extended Temporal Interval Logic (TILCOX) [5]. As an alternative, researchers have investigated using observers to capture real-time specification patterns. Observers are code/model fragments written in the modeling or implementation language to be verified, such as timed automata, timed Petri nets, source code, and Simulink, among others. For example, Gruhn [4] and Abid [11] describe real-time specifications as state machines in timed automata and timed Petri nets, respectively. A benefit of this approach is that rather than checking complex timed temporal logic properties (which can be very expensive and may not be supported by a wide variety of analysis tools), it is possible to check simpler properties over the observer.

Despite this benefit, capturing real-time specification patterns with observers can be challenging, especially in the presence of overlapping "trigger events." That is, if P occurs multiple times before low time units have elapsed in the example above. For example, most of the observers in Abid [11] explicitly are not defined for 'global' scopes, and Gruhn, while stating that global properties are supported, only checks a pattern for the first occurrence of the triggering event in an infinite trace.

In this effort, we examined the use of observers to capture specification patterns that can involve overlapping triggering events. We implemented our work in AGREE [12] which translates Architectural Analysis and Design Language (AADL) [13] specifications into Lustre programs [14][2]. We describe the conditions under which we can use observers to faithfully represent the semantics of patterns, for both positive instances of patterns and negations of patterns. We call the former use properties and the latter use constraints.

The reason that we consider negations of patterns is that our overall goal is to use real-time specification patterns in the service of assume/guarantee compositional reasoning. We used AGREE to verify quasi-synchronous properties for four different models in the Software Productivity Initiative (SWPI) project. In other recent efforts [15, 16], we used the AGREE tool suite for reasoning about discrete time behavioral properties of complex models. Through adding support for Requirements Specification Language (RSL) patterns [17] and calendar automata [18–20], it becomes possible to lift our analysis to real-time systems. In AGREE, we prove implicative

[2] Although our formalisms are expressed as Lustre specifications, the concepts and proofs presented in this report are applicable to many other popular model checking specification languages.

properties: given that subcomponents satisfy their contracts, then a system should satisfy its contract. This means that the RSL patterns for subsystems are used under a negation.

The rest of this report is organized as follows. In Section 2.2 we list formal definitions that are used to describe the semantics of our patterns and perform proofs about our observers. In Section 3 we give a brief overview of the compositional verification rules of AGREE. In Section 3.1 we describe the RSL patterns that we have implemented in AGREE along with their formal semantics. In Section 3.2 we describe in detail two of the examples that we used the RSL patterns to model. In Section 4 we discuss the analysis results for these examples. Overall we make the following contributions with this report:

- We demonstrate a method for translating RSL Patterns into Lustre observers and system invariants.
- We prove that it is possible to efficiently capture patterns involving arbitrary overlapping intervals in Lustre using non-determinism.
- We argue that there is no method to efficiently encode a transition system in Lustre that implements the exact semantics of all of the RSL patterns when considering their negation.
- We demonstrate how to encode these patterns as Lustre constraints for practical systems.
- We discuss the use of these patterns to model two real-world avionics system.

2.1 Related Work

This work focused on implementing real-time patterns from the RSL [6] that was created as part of the Cost-efficient Methods and Processes for Safety-relevant Embedded Systems (CESAR) project [17]. This language is an extension and modularization of the Contract Specification Language (CSL) [24]. The goal of both of these projects was to provide contract-based reasoning for complex embedded systems. We chose this as our initial pattern language because of the similarity in the contract reasoning approach used by our AGREE tool suite [12].

There is considerable work on real-time specification patterns for different temporal logics. Konrad and Cheng [3] provide the first systematic study of real-time specification patterns, adapting and extending the patterns of Dwyer [2] for three different temporal logics: TCTL [8], MTL [9], and RTGIL [10]. Independently, Gruhn [4] constructed a real-time pattern language derived from Dwyer, presenting the patterns as observers in timed automata. In Konrad and Cheng, multiple (and overlapping) occurrences of patterns are defined in a trace, whereas in Gruhn, only the first occurrence of the patterns considered. This choice sidesteps the question of adequacy for overlapping triggering events (as discussed in Section 3.1), but limits the expressiveness of the specification. We use a weaker specification language than Konrad [3] which allows better scaling to our analysis, but we also consider multiple occurrences of patterns, unlike Gruhn [4]. Bellini [5] creates a classification scheme for both Gruhn's and Konrad's patterns and provides a rich temporal language called TILCOX that allows more straightforward expression of many of the real-time patterns. Like [3], this work considers multiple overlapping occurrences of trigger events.

The closest work to ours is probably that of Abid et. al [11], who encode a subset of the CSL patterns as observers in a timed extension of Petri nets called Timed Transition System (TTS), and supplement the observers with properties that involve both safety and liveness in Linear Temporal Logic (LTL). For most of the RSL patterns considered, the patterns are only required to hold for the first triggering event, rather than globally across the input trace. In addition, the use of full LTL makes the analysis more difficult with inductive model checkers. Other recent work [7] considers very expressive real-time contracts with quantification for systems of systems. This quantification makes the language expressive, but difficult to analyze.

Other researchers including Pike [25] and Sorea [26] have explored the idea of restricting traces to disallow overlapping events in order to reason about real-time systems using safety properties. The authors of [27] independently developed a similar technique of using a *trigger* variable to specify real-time properties that quantify over events.

2.2 Definitions

AGREE proves properties of architectural models compositionally by proving a series of lemmas about components at different levels in the model's hierarchy. A detailed description of how these proofs are constructed is provided in [12, 15] and a proof sketch of correctness of these rules is described in [12, 21]. We also give a brief overview of these rules in the following section. The AGREE tool translates AADL models annotated with component assumptions, guarantees, and assertions into Lustre programs. Our explanations and formalizations in this report are described by these target Lustre specifications. Most other SMT-based model checkers use a specification language that has similar expressivity as Lustre; the techniques we present in this report can be applied generally to other model checking specification languages.

A Lustre program $M = (V, T, P)$ can be thought of as a finite collection of named variables V, a transition relation T, and a finite collection of properties P. Each named variable is of type *bool*, *integer*, or *real*. The transition relation is a Boolean constraint over these variables and theory constants; the value of these variables represents the program's current *state*, and the transition relation constrains how the state changes. Each property $p \in P$ is also a Boolean constraint over the variables and theory constants. We sometimes refer to a Lustre program as a model, specification, or transition system. The AGREE constraints specified via assumptions, assertions, or guarantees in an AADL model are translated to either constraints in the transition relation or properties of the Lustre program.

The expression for T contains common arithmetic and logical operations ($+, -, *, \div, \vee, \wedge, \Rightarrow, \neg, =$) as well as the "if-then-else" expression (*ite*) and two temporal operations: $\rightarrow$ and *pre*. The $\rightarrow$ operation evaluates to its left hand side value when the program is in its initial state. Otherwise it evaluates to its right hand side value. For example, the expression: $true \rightarrow false$ is true in the initial state and *false* otherwise. The *pre* operation takes a single expression as an argument and returns the value of this expression in the previous state of the transition system. For example, the expression: $x = (0 \rightarrow pre(x) + 1)$ constrains the current value of variable x to be 0 in the initial state otherwise it is the value of x in the previous state incremented by 1.

In the model's initial state the value of the *pre* operation on any expression is undefined. Every occurrence of a *pre* operator must be in a subexpression of the right hand side of the $\rightarrow$ operator. The *pre* operation can be performed on expressions containing other *pre* operators, but there must be $\rightarrow$ operations between each occurrence of a *pre* operation. For example, the expression: $true \rightarrow pre(pre(x))$ is not well-formed, but the expression: $true \rightarrow pre(x \rightarrow pre(x))$ is well-formed.

A Lustre program models a state transition system. The current values of the program's variables are constrained by values of the program's variables in the previous state. In order to model timed systems, we introduce a real-valued variable t which represents how much time has elapsed during the previous transitions of the system. We adopt a similar model as *timeout automata* as described in [18]. The system that is modeled has a collection of *timeouts* associated with the time of each "interesting event" that will occur in the system. The current value of t is assigned to the least timeout of the system greater than the previous elapsed time. Specifically, Formula 1 shows that t has the following constraint:

$$t = 0 \rightarrow pre(t) + min\_pos(t_1 - pre(t), \ldots, t_n - pre(t)) \qquad (1)$$

where $t_1, \ldots, t_n$ are variables representing the timeout values of the system. The function *min_pos* returns the value of its minimum positive argument. We constrain all the timeouts of the system to be positive. A timeout may also be assigned to positive infinity (∞)[3]. There should always be a timeout that is greater than the current time (and less than ∞). If this is true, then the invariant $true \rightarrow t > pre(t)$ holds for the model, i.e., time always progresses.

A sequence of states is called a *trace*. A *trace* is said to be *admissible* (w.r.t. a Lustre model or transition relation) if each state and its successor satisfy the transition relation. We adopt the common notation (σ, τ) to represent a trace of a timed system where σ is a sequence of states ($\sigma = \sigma_1\sigma_2\sigma_3 \ldots$) and τ is a sequence of time values ($\tau = \tau_1\tau_2\tau_3 \ldots$) such that $\forall i : \tau_i < \tau_{i+1}$. In some literature, state transitions may take place without any time progress (i.e., $\forall i : \tau_i \leq \tau_{i+1}$). We do not allow these transitions as it dramatically increases the complexity of a model's Lustre encoding.

A Lustre program implicitly describes a set of admissible *traces*. Each state σ_n in the sequence represents the value of the variables V in state n. Each time value τ_n represents the value of the time variable t in state n. We use the notation $\sigma_n \models e$, where e is Lustre expression over the variables V and theory constants, if the expression e is satisfied in the state σ_n. Similarly, we use $\sigma_n \not\models e$ when e is not satisfied in the state σ_n. A property p is *true* (or invariant) in a model if and only if for every admissible trace $\forall n : \sigma_n \models p$. For the purposes of this work, we only consider models that do not admit so-called "Zeno traces" [22]. A trace (σ, τ) is a Zeno trace if and only if $\exists v \forall i : \tau_i < v$, i.e., time never progresses beyond a fixed point.

[3] In practice, we allow a timeout to be a negative number to represent infinity. This maintains the correct semantics for the constraint for t in Formula 1.

3. Methods, Assumptions, and Procedures

In this section we briefly describe the rules that AGREE uses to create compositional proofs. A more complete description is in [12] and a proof of correctness of these rules is provided in [12, 21].

AGREE is a language and a tool for compositional verification of AADL models. It is implemented as a plugin to the Open Source AADL Tool Environment (OSATE). The behavior of a model is described by *contracts* specified on each component. A contract contains a set of *assumptions* about the component's inputs and a set of *guarantees* about the component's outputs. The assumptions and guarantees may also contain predicates that reason about how the state of a component evolves over time. By default the state transitions of each component in the model occur synchronously with every other component (i.e., each component runs on the same clock). However, AGREE can also model systems that have components with independent clocks. The user can specify arbitrary constraints for these clocks or they can use built in quasi-synchronous constraints that were developed under the SWPI project.

The guarantees of a component must be true provided that the component's assumptions have always been true. The goal of the analysis is to prove that a component's contract is entailed by the contracts of its subcomponents.

Formally, let a system S: (A, G, C) consist of a set of assumptions A, guarantees G, and subcomponents C. We use the notation S_g to represent the conjunction of all guarantees of S and S_a to represent the conjunction of all assumptions of S. Each subcomponent $c \in C$ is itself a system with assumptions, guarantees, and subcomponents. The goal of AGREE's analysis is to prove that the system's guarantees hold as long as its assumptions have always held. This is accomplished by proving that Formula 2 is an invariant.

$$\mathbf{H}(S_a) \Rightarrow S_g \tag{2}$$

The predicate **H** is true if its argument has held *historically* (i.e., the expression has been *true* at every step up until and including *now*). In order to prove that Formula 2 is invariant, we prove that if the system level assumptions have held historically, and if all the subcomponent contracts have held historically, then the system level guarantees hold. This is described by Formula 3.

$$\mathbf{H}(S_a) \wedge \bigwedge_{c \in C} \mathbf{H}(c_a) \Rightarrow c_g \ \Rightarrow S_g \tag{3}$$

Proofs of systems with multiple levels of hierarchy take place *compositionally*. AGREE attempts to prove Formula 3 for each level of the system. This allows proofs of contracts for components at the highest levels of the system to rely only on the abstract representation of their direct subcomponents provided by their contracts. This method of abstraction allows AGREE to scale to larger systems.

As mentioned earlier, users can specify different clock domains for components in AGREE. Formula 3 assumes that each component executes synchronously. If each component $c \in C$ has a different clock, AGREE instead attempts to prove Formula 4.

$$\mathbf{H}(S_a) \wedge \bigwedge_{c \in C} \mathbf{H}(c_c \Rightarrow c_a \Rightarrow c_g) \Rightarrow s_g \tag{4}$$

Here the variable c_c is used to represent the *clock variable* for component c. The semantics of a component's clock variable is such that whenever c_c is true the component transitions to its next state based on its current state and the constraints specified by its guarantees. However, the components contract is only enforced when its clock ticks.

AGREE uses a syntax similar to Lustre to express a contract's assumptions and guarantees [14]. AGREE translates an AADL model annotated with AGREE annexes into Lustre corresponding to Formula 4 and then queries a user selected model checker. AGREE then translates the results from the model checker back into OSATE so they can be interpreted by the user. For this project we chose to use the JKind model checker [23].

3.1 RSL Patterns and Semantics Overview

For this effort, we chose to target the natural language patterns proposed in the Cost-efficient Methods and Processes for Safety-relevant Embedded Systems project because they are representative of many types of natural language requirements [17]. These patterns are divided into a number of categories. The categories of interest for this work are the *functional patterns* and the *timing patterns*. Some examples of the functional patterns are:

1. **Whenever** event **occurs** event **occurs during** interval
2. **Whenever** event **occurs** condition **holds during** interval
3. **When** condition **holds during** interval event **occurs during** interval
4. **Always** condition

Some examples of timing patterns are:

1. Event **occurs each** period [**with jitter** jitter]
2. Event **occurs sporadic with IAT** interarrivaltime [**and jitter** jitter]

Generally speaking, the timing patterns are used to constrain how often a system is required to respond to events. For instance, a component that listens to messages on a shared bus might assume that new messages arrive at most every 50ms. The second timing pattern listed above would be ideal to express this assumption. In AGREE, this requirement may appear as a system assumption using the pattern shown in Figure 1.

new_message **occurs sporadic with IAT** 50.0

Figure 1: An instance of a timing pattern to represent how frequently a message arrives on a shared bus.

The functional patterns can be used to describe how the system's state changes in response to external stimuli. Continuing with the previous example, suppose that the bus connected component performs some computation whenever a new message arrives. The functional patterns can be used to describe when a thread is scheduled to process this message and how long the thread takes to complete its computation. The intervals in these patterns have a specified lower and upper bound, and they may be open or closed. The time specified by the lower and upper bound corresponds to the time that progresses since the triggering event occurs. Both the lower and upper bounds must be positive real numbers, and the upper bound must be greater than or equal to the lower bound. An AGREE user may specify the instances of patterns shown in Figure 2 as properties she would like to prove about this system. For the purposes of demonstration we assume that the thread should take 10ms to 20ms to execute.

```
Always new_message = thread_start
Whenever thread_start occurs thread_stop occurs during [10.0, 20.0]
```

Figure 2: Two instances of a functional patterns used to describe when a thread begins executing, and how long it takes to execute.

Figure 3 shows a graphical representation of the first functional pattern listed at the beginning of this section. The variable t_c represents the time that event c occurs. Similarly, the variable t_e represents the time that event e occurs. The formal semantics for many of the RSL patterns are described in [6]. The semantics for the pattern described in Figure 3 are represented by the set of admissible traces $\mathcal{L}_{patt}$ described below.

$$\mathcal{L}_{patt} = \{(\sigma, \tau) \mid \forall i \exists j : \sigma_i \models c \Rightarrow (j > i) \land (\tau_i + l \leq \tau_j \leq \tau_i + h) \land (\sigma_j \models e)\}$$

The remainder of this section discusses how the pattern in Figure 3 can be translated into either a Lustre property or a constraint on the admissible traces of a transition system described by Lustre. Although we discuss only this pattern, the techniques that we present can be applied generally to all except one of the functional and timing RSL patterns[4].

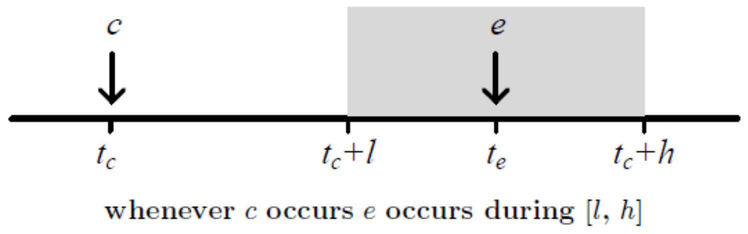


Figure 3: A graphical representation for the RSL pattern.

[4] The single pattern that cannot be implemented requires an independent event to occur for each of an unbounded number of causes. There are 12 functional and timing RSL patterns in total.

3.1.1 Implementing RSL Patterns as Lustre Properties

One can determine if a transition system described in Lustre admits only traces in $\mathcal{L}_{patt}$ by adding additional constraints over fresh variables (variables that are not already present in the program) to the model. This commonly used technique is referred to as adding an observer to the model. These constraints are over fresh variables: *run*, *timer*, rec_c, and *pass*; they are shown in Figure 4. The constraints only restrict the values of the fresh variables, therefore they do not restrict the traces admissible by the transition relation.

$$\begin{array}{ll}
1. & run = (rec_c \rightarrow ite(pre(run) \wedge e \wedge l \leq timer \leq h, \\
 & \qquad\qquad false, \\
 & \qquad\qquad ite(rec_c, true, pre(run)))) \\
2. & timer = (0 \rightarrow ite(pre(run), pre(timer) + (t - pre(t)), 0)) \\
3. & rec_c \Rightarrow c \\
4. & pass = (timer \leq h)
\end{array}$$

Figure 4: The constraints added to a transition relation to verify if only the traces of $\mathcal{L}_{patt}$ are admissible. The transition relation only admits traces of $\mathcal{L}_{patt}$ if and only if the variable *pass* is invariant.

The intuition behind these constraints is that one can record how much time progresses since an occurrence of c. This time is recorded in the *timer* variable. The value of the timer variable only increases if the previous value of the *run* variable is *true*. The *run* variable is true if an occurrence of c is recorded and no occurrence of e happens until after the timer counts to at least l. The variable rec_c non-deterministically records an occurrence of c. If the transition system admits a trace outside of $\mathcal{L}_{patt}$, then the rec_c variable can choose to record only an event that violates the conditions of $\mathcal{L}_{patt}$. In this case the *pass* variable will become *false* in some state.

Theorem 1. *Let $\mathcal{L}_M$ represent the admissible traces of a transition system containing the constraints of Figure 4. The transition system admits only traces in $\mathcal{L}_{patt}$ if and only if the property pass is invariant. Formally: $(\mathcal{L}_M \subseteq \mathcal{L}_{patt}) \Leftrightarrow (\forall \sigma, \tau, i : (\sigma, \tau) \in \mathcal{L}_M \Rightarrow \sigma_i \models pass)$*

Proof: First we show that if *pass* is invariant for a trace of the transition relation, then that trace is in $\mathcal{L}_{patt}$.

Lemma 1. $(\forall \sigma, \tau, i : (\sigma, \tau) \in \mathcal{L}_M \Rightarrow \sigma_i \models pass) \Rightarrow (\mathcal{L}_M \subseteq \mathcal{L}_{patt})$.

Proof: Towards contradiction, assume $\mathcal{L}_M \not\subseteq \mathcal{L}_{patt}$. Let (σ, τ) be a trace in $\mathcal{L}_M$ but not in $\mathcal{L}_{patt}$. Since $(\sigma, \tau) \notin \mathcal{L}_{patt}$, by definition there exists i such that $\sigma_i \models c$ and

$$\forall j : (j > i) \wedge \tau_i + l \leq \tau_j \leq \tau_i + h \Rightarrow \sigma_j \not\models e. \tag{5}$$

Without loss of generality, we can assume that this is the only time when c is recorded. That is, $\sigma_i \models rec_c$ and $\forall k : k \neq i \Rightarrow \sigma_k \not\models rec_c$. From constraint 1 in Figure 4 we have:

$$\forall j : ((j < i) \Rightarrow \sigma_j \not\models run) \wedge ((\tau_i \leq \tau_j < \tau_i + l) \Rightarrow \sigma_j \models run)$$

This can actually be strengthened more. From Formula 5 the event e does not occur between $\tau_i + l$ and $\tau_i + h$. So the variable run will become invariant after τ_i.

$$\forall j : ((j < i) \Rightarrow \sigma_j \not\models \text{run}) \wedge (\tau_i \leq \tau_j) \Rightarrow \sigma_j \models run)$$

From this and constraint 2 in Figure 4, we have

$$\forall j : (j \leq i) \Rightarrow \sigma_j \models timer = 0$$

and

$$\forall j : (\tau_i < \tau_j) \Rightarrow (\sigma_j \models timer = (pre(timer) + (\tau_j - \tau_{j-1})))$$

From this and the invariant $\forall i : \tau_{i+1} > \tau_i$, we have

$$\forall j : (\tau_i < \tau_j) \Rightarrow (\sigma_j \models timer > pre(timer))$$

Therefore since the value of timer is zero before τ_i and always increasing after τ_i, and since we only consider non-Zeno traces ($\forall v \exists i : v < \tau_i$), eventually $timer > h$ and so *pass* becomes *false*. This contradicts the assumption ($\forall \sigma, \tau, i : (\sigma, \tau) \in \mathcal{L}_M \Rightarrow \sigma_i \models pass$). Therefore $\mathcal{L}_M \subseteq \mathcal{L}_{patt}$.

Next we show if a *trace* of $\mathcal{L}_M$ is in $\mathcal{L}_{patt}$, then *pass* is invariant for this *trace*.

Lemma 2. $(\mathcal{L}_M \subseteq \mathcal{L}_{patt}) \Rightarrow (\forall \sigma, \tau, i : (\sigma, \tau) \in \mathcal{L}_M \Rightarrow \sigma_i \models pass)$

Proof. Towards contradiction, assume that there exists a trace of $\mathcal{L}_M$ for which pass is not invariant. This means that for some state $\sigma_j \models timer > h$. For this to be *true*, the timer must be running continuously since it started with some recorded occurrence of c. That is there exists i such that $\sigma_i \models timer = 0$, $\sigma_i \models rec_c$, $\sigma_i \models c$, $\forall k : i \leq k < j \Rightarrow \sigma_k \models run$, and $\tau_j - \tau_i > h$. Thus $\forall k : i \leq k \leq j \Rightarrow \sigma_k \models timer = \tau_k - \tau_i$. By the definition of $\mathcal{L}_{patt}$ we have a k such that $\tau_i + l \leq \tau_k \leq \tau_i + h$ and $\sigma_k \models e$. This means $l \leq \tau_k - \tau_i \leq h$ and so $\sigma_k \models l \leq timer \leq h$. Therefore $\sigma_k \models run$. We also have $\tau_k \leq \tau_i + h < \tau_j$ so that $k < j$. Thus from $\forall k : i \leq k < j \Rightarrow \sigma_k \models run$ we have $\sigma_k \models run$ which is a contradiction. Therefore, *pass* is invariant.

From Lemmas 1 and 2 we have $(\mathcal{L}_M \subseteq \mathcal{L}_{patt}) \Leftrightarrow (\forall \sigma, \tau, i : (\sigma, \tau) \in \mathcal{L}_M \Rightarrow \sigma_i \models pass)$.

3.1.2 Implementing RSL Patterns as Lustre Constraints

As we demonstrated with Figure 4, one can specify a Lustre property that verifies whether or not some transition system only admits traces of $\mathcal{L}_{patt}$. However, it is surprisingly non-trivial to actually implement a transition system that admits exactly the traces of $\mathcal{L}_{patt}$. Naively, one could attempt to add the constraints of Figure 4 to a transition system and then assert that pass is invariant. However, this transition system will admit all traces where every occurrence of c is never recorded ($\forall \sigma_i : \sigma_i \models rec_c$). Clearly some of these traces would not be in $\mathcal{L}_{patt}$.

We conjecture that given the Lustre expression language described in Section 2.2 it is not possible to model a transition system that admits only and all of the traces of $\mathcal{L}_{patt}$. The intuition behind this claim is that Lustre specifications contain a fixed number of state variables, and variables have non-recursive types. Thus a Lustre specification only has a finite amount of memory (though it can, for example, have arbitrary sized integers). If a Lustre specification has n variables we can always consider a trace in $\mathcal{L}_{patt}$ where event c occurs more than n times in a tiny interval. In order for the pattern to hold true, the Lustre specification must constrain itself so that at least one occurrence of e occurs precisely between $t_c + l$ and $t_c + h$ after each event c. This requires "more memory" than the Lustre specification has available.

Rather than model the exact semantics of this pattern, we choose to take a more pragmatic approach. We model a strengthened version of Figure 3 which does not allow overlapping instances of the pattern. That is, after an event c there can be no more occurrences of c until the corresponding occurrence of e. We do this by proving that c cannot occur frequently enough to cause an overlapping occurrence of the pattern. Then if we constrain the system based on a simple non-overlapping check of the pattern, the resulting system is the same as if we had constrained it using the full pattern. This simple non-overlapping check and the property limiting the frequency of c are both easily expressed in Lustre since they only look back at the most recent occurrence of c. Moreover, they can both be used freely in positive and negative contexts. Formally, the property we prove is $\mathcal{L}_{prop}$ and the constraints we make are $\mathcal{L}_{cons}$:

$$\mathcal{L}_{prop} = \{(\sigma, \tau) \mid \forall i : \sigma_i \models c \Rightarrow \forall j : (j > i) \wedge (\tau_j \leq \tau_i + h) \wedge \sigma_j \models c \Rightarrow$$

$$\exists k \in (i, j] : \tau_i + l \leq \tau_k \wedge \sigma_k \models e\}$$

$$\mathcal{L}_{cons} = \{(\sigma, \tau) \mid \forall i : \sigma_i \models c \Rightarrow$$

$$\exists j : (j > i) \wedge [(\tau_i + l \leq \tau_j \leq \tau_i + h \wedge \sigma_j \models e) \vee (\tau_j \leq \tau_i + h \wedge \sigma_j \models c)]\}$$

The correctness of $\mathcal{L}_{prop}$ and $\mathcal{L}_{cons}$ are captured by the following theorem.

Theorem 2. Let M be a transition system and $\mathcal{L}_M$ its corresponding set of admissible traces. Suppose $\mathcal{L}_M \subseteq L_{prop}$. Then $\mathcal{L}_{cons}$ and $\mathcal{L}_{patt}$ are equivalent restrictions on $\mathcal{L}_M$, that is $\mathcal{L}_M \cap \mathcal{L}_{cons} = \mathcal{L}_M \cap \mathcal{L}_{patt}$.

Proof. We prove the theorem by showing that the subset relationship between $\mathcal{L}_M \cap \mathcal{L}_{cons}$ and $\mathcal{L}_M \cap \mathcal{L}_{patt}$ holds in both directions.

Lemma 3. $\mathcal{L}_M \cap \mathcal{L}_{patt} \subseteq \mathcal{L}_M \cap \mathcal{L}_{cons}$

Proof. From the definitions of $\mathcal{L}_{patt}$ and $\mathcal{L}_{cons}$ it follows directly that $\mathcal{L}_{patt} \subseteq \mathcal{L}_{cons}$. Therefore $\mathcal{L}_M \cap \mathcal{L}_{patt} \subseteq \mathcal{L}_M \cap \mathcal{L}_{cons}$.

Lemma 4. Suppose $\mathcal{L}_M \subseteq \mathcal{L}_{prop}$, then $\mathcal{L}_M \cap \mathcal{L}_{cons} \subseteq \mathcal{L}_M \cap \mathcal{L}_{patt}$

Proof. Suppose towards contradiction that $\mathcal{L}_M \cap \mathcal{L}_{cons} \nsubseteq \mathcal{L}_M \cap \mathcal{L}_{patt}$. Consider a trace $(\sigma, \tau) \in \mathcal{L}_M \cap \mathcal{L}_{cons}$ with $(\sigma, \tau) \notin \mathcal{L}_M \cap \mathcal{L}_{patt}$. Then we have $(\sigma, \tau) \in \mathcal{L}_{cons}$, $(\sigma, \tau) \in \mathcal{L}_{prop}$, and $(\sigma, \tau) \notin \mathcal{L}_{patt}$. From the definition of $\mathcal{L}_{patt}$ we have an i such that $\sigma_i \models c$ and

$$\forall j : (j > i) \wedge (\tau_i + l \le \tau_j \le \tau_i + h) \Rightarrow \sigma_j \models e. \tag{6}$$

Then from the definition of $\mathcal{L}_{cons}$ with $\sigma_i \models c$ we have a j such that $j > i$ and either $(\tau_i + l \le \tau_j \le \tau_i + h \wedge \sigma_j \models e)$ or $(\tau_j \le \tau_i + h \wedge \sigma_j \models c)$. The former option directly contradicts Formula 6, so we must have $\tau_j \le \tau_i + h$ and $\sigma_j \models c$. From the definition of $\mathcal{L}_{prop}$ with $\sigma_i \models c$ and our j, we have a k in $(i, j]$ such that $\tau_i + l \le \tau_k$ and $\sigma_k \models e$. From $k \le j$ we have $\tau_k \le \tau_j$ and thus $\tau_i + l \le \tau_k \le \tau_i + h$. Instantiating Formula 6 with k yields $\sigma_k \models e$, a contradiction. Therefore $\mathcal{L}_M \cap \mathcal{L}_{cons} \subseteq \mathcal{L}_M \cap \mathcal{L}_{patt}$.

From Lemmas 3 and 4 we have $\mathcal{L}_M \cap \mathcal{L}_{cons} = \mathcal{L}_M \cap \mathcal{L}_{patt}$.

Example 1. Suppose we want to model a system of components communicating on a shared bus. The transition relation for this system must contain constraints that dictate when threads can start and stop and how frequently new messages may arrive. First we constrain the event new message from occurring too frequently according to the pattern instance in Figure 1. Let $\mathcal{L}_{nm}$ represent the set of admissible traces for this pattern. This set is defined explicitly in Formula 1.

$$\mathcal{L}_{nm} = \{(\sigma, \tau) \mid \forall i : \sigma_i \models new\_message \Rightarrow \neg[\exists j : (j > i) \wedge (\tau_j < \tau_i + 50) \wedge (\sigma_j \models new\_message)]\}$$

Suppose we wish to constrain the system to the pattern instances in Figure 2. The first pattern instance is represented by the set $\mathcal{L}_{start}$ and the second by $\mathcal{L}_{stop}$:

$$\mathcal{L}_{start} = \{(\sigma, \tau) \mid \forall i : \sigma_i \models new\_message \Rightarrow \sigma_i \models thread\_start\}$$

$$\mathcal{L}_{stop} = \{(\sigma, \tau) \mid \forall i \exists j : \sigma_i \models thread\_start \Rightarrow (j > i) \wedge (\tau_i + l \le \tau_j \le \tau_i + h) \wedge (\sigma_j \models thread\_stop)\}$$

Let $\mathcal{L}_M$ denote the admissible traces of the transition system that is being modeled. The goal is to specify the transition system in Lustre such that $\mathcal{L}_M = \mathcal{L}_{nm} \cap \mathcal{L}_{start} \cap \mathcal{L}_{stop}$. Writing a Lustre constraint to represent the set of traces $\mathcal{L}_{start}$ is trivial. The traces that are contained in $\mathcal{L}_{start}$ are those whose states all satisfy the expression $new\_message = thread\_stop$. However, as we noted earlier, it is not possible to develop a set of Lustre constraints that admit only (and all of) the traces of $\mathcal{L}_{stop}$.

Note that the second pattern in Figure 2 is an instance of the pattern described in Figure 3. Therefore we can split the set $\mathcal{L}_{stop}$ into two sets, $\mathcal{L}_{stopc}$ and $\mathcal{L}_{stopp}$:

$$\begin{aligned}\mathcal{L}_{stopc} = \{(\sigma, \tau) \mid \forall i : \sigma_i \models thread\_start \Rightarrow \exists j : (j > i) \wedge \\ [(\tau_i + l \le \tau_j \le \tau_i + h \wedge \sigma_j \models thread\_stop) \vee \\ (\tau_j \le \tau_i + h \wedge \sigma_j \models thread\_start)]\}\end{aligned}$$

$$\mathcal{L}_{stopp} = \{(\sigma, \tau) \mid \forall i : \sigma_i \models thread\_start \Rightarrow \forall j : (j > i) \wedge$$

$(\tau_j \leq \tau_i + h) \wedge \sigma_j \models \mathit{thread\_start} \Rightarrow$

$\exists k \in (i, j] : \tau_i + l \leq \tau_k \wedge \sigma_k \models \mathit{thread\_stop}\}$

In this example, the sets of admissible traces representing the patterns happen to have the following relationship:

$$\mathcal{L}_{nm} \cap \mathcal{L}_{start} \subseteq \mathcal{L}_{stopp} \tag{7}$$

This is because for every trace in $\mathcal{L}_{nm}$ the event *new_message* only occurs at most every 50ms. Likewise, for each state of every trace of $\mathcal{L}_{start}$ the variable *thread_start* is true if and only if *new_message* is true. Finally, the set $\mathcal{L}_{stopp}$ contains every trace where *thread_start* occurs at most every 20ms. From Formula 7 and Theorem 2 we have $\mathcal{L}_{nm} \cap \mathcal{L}_{start} \cap \mathcal{L}_{stopc} = \mathcal{L}_{nm} \cap \mathcal{L}_{start} \cap \mathcal{L}_{stop}$. Thus the system $\mathcal{L}_{nm} \cap \mathcal{L}_{start} \cap \mathcal{L}_{stopc}$, which we can model in Lustre, is equivalent to a system constrained by the pattern instances in Figures 1 and 2.

Example 1 is meant to demonstrate that, in practical systems, there is usually some constraint on how frequently events outside the system may occur. Systems described by the functional RSL patterns generally have some limitations on how many events they can respond to within a finite amount of time. The Lustre implementations of $\mathcal{L}_{cons}$ and $\mathcal{L}_{prop}$ are simpler than Figure 4, and their proof of correctness is also simpler then Theorem 1, though we omit both due to space limitations.

3.1.3 RSL Patterns Implemented in AGREE

In this section we list all of the RSL patterns that we have implemented in AGREE along with their formal semantics, Lustre property observer, and Lustre constraint observer. The English language description for each pattern is taken nearly verbatim from the CESAR documentation titled “Definition and exemplification of RSL and RMM” [1].

In each of the following figures we first list the syntax for the pattern that AGREE accepts. Immediately afterwards we define the set of traces that the pattern accepts. This set is denoted by the symbol $\mathcal{L}$ for each pattern. We then list the Lustre observers that AGREE generates to implement the pattern. The first observer listed is labelled as the *property observer*. AGREE generates this observer when the pattern is used in a positive context. This occurs when a pattern is used in a system level guarantee that we are proving or a subcomponent assumption that we are proving. The *constraint observer* is used when the pattern appears in a negative context. This is either when a pattern is being used to describe a subcomponent guarantee, a system level assumption, or a component assertion. The *constraint property* is created whenever the constraint observer is used. If the *pass* variable is invariant then the constraint observer property implements the semantics of the pattern. The reason for generating the constraint property observers is described in the previous section. For example, the constraint property described in Figure 5 implements $\mathcal{L}_{prop}$ as described in Example 1 in the previous section.

3.1.3.1 Time Functions

When AGREE translates each pattern into an observer it introduces extra variables into the Lustre program. Often the model checker needs to discover lemmas containing these new variables in order to prove the properties of interest. These lemmas can be very subtle, and the model checker may never discover them. In order to help the model checker the user may specify lemmas by hand for the model checker to verify and use to prove other properties. However, if the user is unable to reference these new variables created by the pattern observers they may be unable to specify the lemmas necessary to prove the properties of interest.

To enable the users to specify these lemmas we have introduced three new functions into the AGREE grammar that allow users to reference some of these observer variables in an intuitive way. These functions along with their descriptions are listed below.

$$\boxed{\text{Whenever } c \text{ occurs } e \text{ occurs during } [l,h]}$$

$$\mathcal{L} = \{(\sigma, \tau) \mid \forall i \exists j : \sigma_i \models c \Rightarrow (j > i) \wedge (\tau_i + l \leq \tau_j \leq \tau_i + h) \wedge (\sigma_j \models e)\}$$

Property observer:

$$\begin{array}{ll}
1. & run = (rec_c \rightarrow ite(pre(run) \wedge e \wedge l \leq timer \leq h, \\
 & \qquad false, \\
 & \qquad ite(rec_c, true, pre(run)))) \\
2. & timer = (0 \rightarrow ite(pre(run), pre(timer) + (t - pre(t)), 0)) \\
3. & rec_c \Rightarrow c \\
4. & pass = (timer \leq h)
\end{array}$$

Constraint observer:

$$\begin{array}{ll}
1. & timeout = ite(c, range, -1.0 \rightarrow pre(timeout)) \\
2. & l \leq range - t \leq h \\
3. & timeout = t \Rightarrow e
\end{array}$$

Constraint property:

$$\begin{array}{ll}
1. & t_c = ite(c, t, -1.0 \rightarrow pre(t_c)) \\
2. & t_e = ite(e, t, -1.0 \rightarrow pre(t_e)) \\
3. & new_c = (false \rightarrow t_c \neq pre(t_c) \wedge 0.0 \leq pre(t_c)) \\
4. & pass = (true \rightarrow new_c \Rightarrow pre(t_c) + l \leq t_e)
\end{array}$$

Figure 5: If *c* occurs then *e* occurs at least once during the specified interval. If *c* occurs multiple times in close succession it is not necessary that there is an individual *e* that corresponds to each *c* [1].

- **timeof**(*e*) : $\mathbb{B} \rightarrow \mathbb{R}$ - This function evaluates to -1.0 if *e* has never been true in the past. Otherwise it evaluates to the last time that *e* was true. When an AGREE contract is translated to Lustre occurrences of **timeof**(*e*) are replaced by the variable t_e as defined by the observers in Figures 5, 6, 8, and 9.

- **timerise**(e) : $\mathbb{B} \rightarrow \mathbb{R}$ - This function evaluates to -1.0 if e has never been true in the past. Otherwise it evaluates to the last time that e transitioned from false to true. When an AGREE contract is translated to Lustre occurrences of **timerise**(e) are replaced by the variable t_{rise_e} as defined by the observer in Figure 8.

- **timefall**(e) : $\mathbb{B} \rightarrow \mathbb{R}$ - This function evaluates to -1.0 if e has never been false in the past. Otherwise it evaluates to the last time that e was transitioned from true to false. When an AGREE contract is translated to Lustre occurrences of **timefall**(e) are replaced by the variable t_{fall_e} as defined by the observer in Figure 8.

Through the course of this project we discovered that we often need to specify lemmas that constrained the distance between the timing of multiple events. For example, if event e_1 always occurs within s seconds of event e_2 one might specify the lemma: **timeof**(e_1) − **timeof**(e_2) ≤ s. We discuss some of the lemmas we needed to prove in the examples in Section 3.2.

$$\boxed{\textbf{Whenever } c \textbf{ occurs } e \textbf{ holds during } [l,h]}$$

$$\mathcal{L} = \{(\sigma, \tau) \mid \forall i, j : \sigma_i \models c \Rightarrow (j > i) \wedge (\tau_i + l \leq \tau_j \leq \tau_i + h) \Rightarrow (\sigma_j \models e)\}$$

Property observer:

1. $rec_c \Rightarrow c$
2. $t_c = ite(rec_c, t, -1.0 \rightarrow pre(t_c))$
3. $pass = (0 \leq t_c \Rightarrow t_c + l \leq t \leq t_c + h \Rightarrow e)$

Constraint observer:

1. $t_c = ite(c, t, -1.0 \rightarrow pre(t_c))$
2. $0 \leq t_c \Rightarrow t_c + l \leq t \leq t_c + h \Rightarrow e$
3. $timeout = ite(c, t_c + l, -1.0)$

Constraint property:

1. $t_c = ite(c, t, -1.0 \rightarrow pre(t_c))$
2. $new_c = (false \rightarrow t_c \neq pre(t_c) \wedge 0.0 \leq pre(t_c))$
3. $pass = (true \rightarrow new_c \Rightarrow pre(t_c) + h \leq t_c)$

Figure 6: If c occurs then e remains *true* during specified interval [1].

$$\boxed{\textbf{condition } c \textbf{ occurs sporadic with IAT } t \textbf{ [and jitter } j\textbf{]}}$$

$$\mathcal{L} = \{(\sigma, \tau) \mid \forall i : \sigma_i \models c \Rightarrow \forall j : \tau_i \leq \tau_j \leq \tau_i + t \Rightarrow \sigma_j \models c\}$$

Constraint observer:

1. $0.0 \leq next_c \leq p \to true$
2. $-j \leq jitter \leq j$
3. $true \to ite(pre(next_c + jitter = t), pre(next_c) + p \leq next_c, pre(next_c))$
4. $timeout = next_c + jitter$
5. $c = (t = timeout)$

Figure 7: The condition *c* occurs sporadically with inter arrival time *t* and jitter *j*. If this pattern is specified without a value for jitter it is equivalent to using the same pattern with *j* = 0 [1].

When c holds during $[l_1,h_1]$ e occurs during $[l_2,h_2]$

$$\mathcal{L} = \{(\sigma, \tau) \mid \exists i \forall j : \tau_i \leq \tau_j \leq \tau_i + h_1 - l_1 \land \sigma_j \models c \Rightarrow$$
$$\exists k : (k > i) \land (\tau_i + h_1 - l_1 + l_2 \leq \tau_k \leq \tau_i + h_1 - l_1 + h_2) \land (\sigma_k \models e)\}$$

Property observer:

1. $run = (rec_c \rightarrow ite(pre(run) \land e \land l \leq timer \leq h,$
   $false,$
   $ite(rec_c, true, pre(run))))$
2. $timer = (0 \rightarrow ite(pre(run), pre(timer) + (t - pre(t)), 0))$
3. $rec_c \Rightarrow held_c$
4. $pass = (timer \leq h)$
5. $t_{rise_c} = ite(rise(c), t, -1.0 \rightarrow pre(t_{rise_c}))$
6. $t_{fall_c} = ite(fall(c), t, -1.0 \rightarrow pre(t_{fall_c}))$
7. $held_c = (t_{fall_c} < t_{rise_c} \land t_{rise_c} - t_{fall_c} = h - l)$

Constraint observer:

1. $t_{rise_c} = ite(rise(c), t, -1.0 \rightarrow pre(t_{rise_c}))$
2. $t_{fall_c} = ite(fall(c), t, -1.0 \rightarrow pre(t_{fall_c}))$
3. $held_c = (t_{fall_c} < t_{rise_c} \land t_{rise_c} - t_{fall_c} = h - l)$
4. $l \leq range - t \leq h$
5. $timeout_2 = ite(held_c, range, -1.0 \rightarrow pre(timeout_2))$
6. $timeout_2 = t \Rightarrow e$

Constraint property:

1. $t_{rise_c} = ite(rise(c), t, -1.0 \rightarrow pre(t_{rise_c}))$
2. $t_{fall_c} = ite(fall(c), t, -1.0 \rightarrow pre(t_{fall_c}))$
3. $held_c = (t_{fall_c} < t_{rise_c} \land t_{rise_c} - t_{fall_c} = h - l)$
4. $timeout_1 = ite(rise(c), t + (h - l), -1.0 \rightarrow pre(timeout_1))$
5. $t_{held_c} = ite(held_c, t, -1.0 \rightarrow pre(t_{held_c}))$
6. $t_e = ite(e, t, -1.0 \rightarrow pre(t_e))$
7. $new_{held_c} = (false \rightarrow t_{held_c} \neq pre(t_{held_c}) \land 0.0 \leq pre(t_{held_c}))$
8. $pass = (true \rightarrow new_{held_c} \Rightarrow pre(t_{held_c}) + l \leq t_e)$

Figure 8: If the c remains *true* during the first interval e is *true* sometime during the second interval [1].

$\boxed{\textbf{Whenever } c \textbf{ occurs } e_1 \textbf{ implies } e_2 \textbf{ during } [l,h]}$

$$\mathcal{L} = \{(\sigma, \tau) \mid \forall i, j : \sigma_i \models c \Rightarrow (j > i) \land (\tau_i + l \le \tau_j \le \tau_i + h) \Rightarrow \sigma_j \models e_1 \Rightarrow \sigma_j \models e_2\}$$

Property observer:

1. $rec_c \Rightarrow c$
2. $t_c = ite(rec_c, t, -1.0 \rightarrow pre(t_c))$
3. $pass = (0 \le t_c \Rightarrow t_c + l \le t \le t_c + h \Rightarrow e_1 \Rightarrow e_2)$

Constraint observer:

1. $t_c = ite(c, t, -1.0 \rightarrow pre(t_c))$
2. $0 \le t_c \Rightarrow t_c + l \le t \le t_c + h \Rightarrow e_1 \Rightarrow e_2$
3. $timeout = ite(c, t_c + l, -1.0)$

Constraint property:

1. $t_c = ite(c, t, -1.0 \rightarrow pre(t_c))$
2. $new_c = (false \rightarrow t_c \ne pre(t_c) \land 0.0 \le pre(t_c))$
3. $pass = (true \rightarrow new_c \Rightarrow pre(t_c) + h \le t_c)$

Figure 9: If *c* occurs then *e* remains *true* during specified interval [1].

$\boxed{\textbf{condition } c \textbf{ occurs each } p\ [\textbf{with jitter } j]}$

$$\mathcal{L} = \{(\sigma, \tau) \mid \exists i : 0 \le \tau_i \le p \land \forall a : 1 \le a \Rightarrow \exists m : (a-1)p\tau_i - j \le \tau_m \le ap\tau_i + j \land \sigma_m \models c$$
$$\land \forall n : (a-1)p\tau_i - j \le \tau_n \le ap\tau_i + j \land n = m \Rightarrow \sigma_n \models c\}$$

Constraint observer:

1. $0.0 \le next_c \le p \rightarrow true$
2. $true \rightarrow next_c = (pre(next_c) + ite(pre(c), p, 0.0))$
3. $-j \le jitter \le j$
4. $timeout = next_c + jitter$
5. $c = (t = timeout)$

Figure 10: The condition *c* occurs periodically with period *p* and jitter *j*. If this pattern is specified without a value for jitter it is equivalent to using the same pattern with *j* = 0 [1].

3.2 Examples

In this section we discuss our experience re-implementing two of the models from the SWPI project using the real-time patterns that we implemented in AGREE. We make the following assumptions about these systems to ensure that our models are accurate:

1. The only possible system faults are modeled explicitly. E.g., a component cannot fail unless there are constraints in the model which represent component failure.
2. The only communication channels in the system are explicit in the model.
3. The system obeys all system level assumptions explicitly stated in the top level contract of the model.

3.2.1 Wheel Braking System

The Wheel Braking System (WBS) model was derived from an accident report for an Airbus A320 aircraft that occurred on May 21st 1998. The accident occurred because of simultaneous failures in two of the aircrafts braking system. The architecture of the primary Braking System Control Unit (BSCU) is shown in Figure 11.

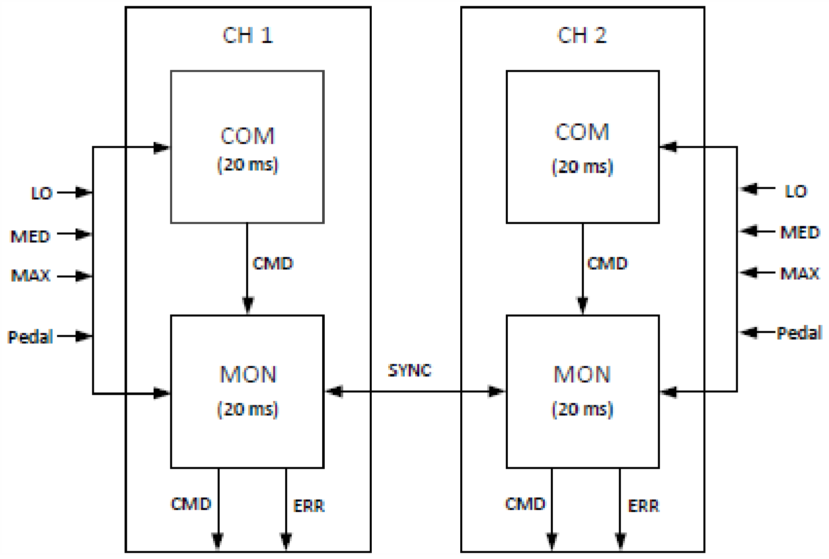


Figure 11: The architecture for the Wheel Braking System.

The BSCU is composed of two channels (CH1 and CH2). Each channel contains one command (COM) and monitor (MON) component. The MON function checks the correctness of the value sent by the COM function. When a disagreement is detected (different result between the COM and MON elements), the MON function raises an error signal. Then, the BSCU switches to the other channel. If this second channel later encounters a disagreement between COM and MON functions, the alternate braking mode is also lost and the only available braking is that provided by manual operation of the parking mode.

Either BSCU channel can operate three different auto-braking modes:

- **LOW** minimum pressure is applied when landing approximately 4 seconds after the ground spoilers are deployed to give a nominal deceleration of 1.7m/s^2 or about 0.17g.
- **MED** medium pressure is applied when landing about 2 seconds after the ground spoilers are deployed to give a deceleration of approximately 3m/s^2.
- **MAX** high pressure is applied as soon as the ground spoilers are deployed give a higher deceleration rate consistent with a rejected take-off or similar takeoff or landing situation.

Because the MON and COM components run on separate clocks each channel can produce spurious errors if the brake commands and/or pedal commands change too frequently. Specifically, the incoming data (either button presses or pedal pressure) must remain constant for a certain amount of time to be guaranteed to be seen by both the MON and COM components (which will guarantee that the MON component does not produce an error).

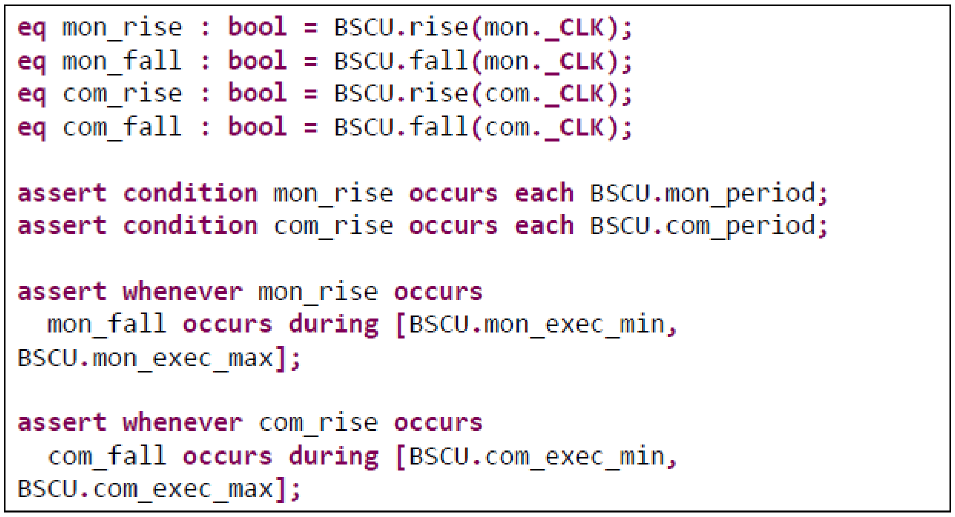
3.2.1.1 Differences between the CRP model and SWPI model

The goal of the CRP project is to develop more concise patterns to represent timing constraints for systems. In order to demonstrate that these patterns serve this purpose, we re-implemented the model developed under the SWPI program to use the newly developed patterns. There are three main differences between the SWPI version of the BSCU and the CRP version:

1. The error logic is slightly different between the two models. In the SWPI version the MON component does not produce an error unless it disagrees with the COM component for a number of clock cycles. In the CRP version it reports an error immediately if it disagrees. There is not really a technical reason for this difference. It is not clear from the documentation what the exact behavior of the real system is.
2. The SWPI version uses quasi-synchronous (QS) constraints to model the timing behavior of the clocks of the MON and COM components. These constraints are not as explicit as constraining the periods and execution times of the individual components. It is more difficult to prove timing properties of the system with QS constraints (because timing is in reference to some base clock). In the CRP version of the model we utilize the real-time patterns to express the period of the component clocks.
3. The SWPI model is not compositional. The model is a flat hierarchy of four components with four clocks. In contrast, the CRP model consists of two channels, and each channel consists of two components. The SWPI model is not organized hierarchically because it is difficult to analyze QS systems this way. Verifying the system compositionally can greatly improve the scalability of model checking.

3.2.1.2 Use of CRP patterns

We make use of real-time patterns in two aspects of the model. First, we use the patterns to constrain how frequently the MON and COM components execute and for how long they execute. These constraints appear as assertions in the CHANNEL component implementation. They are shown in Figure 12:



```
eq mon_rise : bool = BSCU.rise(mon._CLK);
eq mon_fall : bool = BSCU.fall(mon._CLK);
eq com_rise : bool = BSCU.rise(com._CLK);
eq com_fall : bool = BSCU.fall(com._CLK);

assert condition mon_rise occurs each BSCU.mon_period;
assert condition com_rise occurs each BSCU.com_period;

assert whenever mon_rise occurs
  mon_fall occurs during [BSCU.mon_exec_min,
BSCU.mon_exec_max];

assert whenever com_rise occurs
  com_fall occurs during [BSCU.com_exec_min,
BSCU.com_exec_max];
```

Figure 12: Patterns used to constrain the execution of the MON and COM components for each channel of the CRP version of the Wheel Braking System.

We use the constants *mon_period*, *com_period*, *mon_exec_min*, *mon_exec_max*, *com_exec_min*, and *com_exec_max* defined in the BSCU package AGREE annex to set the various parameters for the system. The periods, as indicated in the accident report, are set to 20ms. We do not have values for the minimum and maximum execution times for the components so we made-up values that we believe are plausible (5ms for the minimum and 7ms for the maximum).

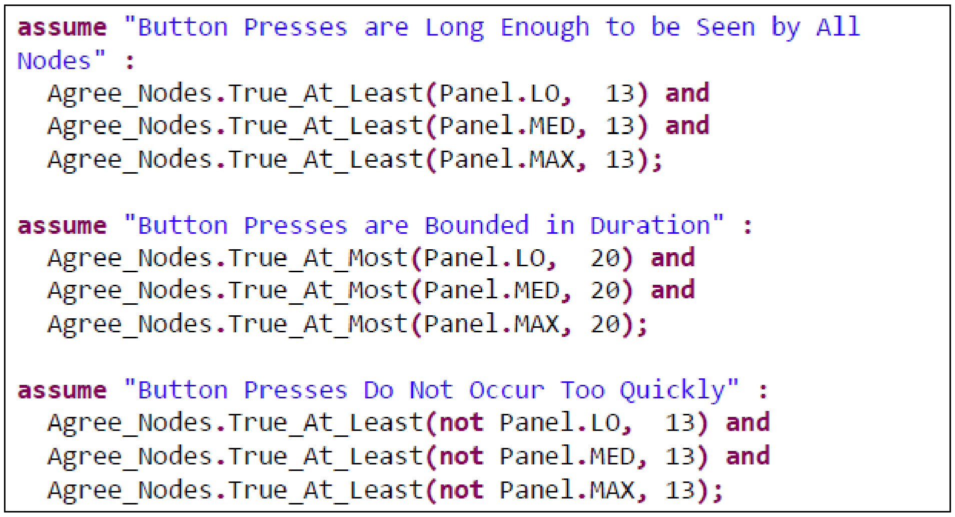
We also used the patterns as system level assumptions to constrain how frequently the buttons are allowed to be pressed and how frequently the pedal value is allowed to change. These assumptions are shown in Figure 13. The constants *button_push_min* and *pedal_push_min* were chosen to satisfy the system level properties. These values were derived analytically based on the periods of the COM and MON components. If these values were to be increased, the system level properties would still be satisfied.

Approved for Public Release; Distribution Unlimited.

```
assume "no simultaneous buttons" :
  not ((Panel.MAX or Panel.MED)
  and (Panel.MAX or Panel.LO)
  and (Panel.MED or Panel.LO));

eq max_rise : bool = BSCU.rise(Panel.MAX);
eq panel_max : bool = Panel.MAX;
assume "no rapid button pushes for Panel.MAX" :
  whenever  max_rise occurs panel_max holds during [0.0,
BSCU.button_push_min];

eq med_rise : bool = BSCU.rise(Panel.MED);
eq panel_med : bool = Panel.MED;
assume "no rapid button pushes for Panel.MED" :
  whenever med_rise occurs panel_med holds during [0.0,
BSCU.button_push_min];

eq lo_rise : bool = BSCU.rise(Panel.LO);
eq panel_lo : bool = Panel.LO;
assume "no rapid button pushes for Panel.LO" :
  whenever lo_rise occurs panel_lo holds during [0.0,
BSCU.button_push_min];

eq pedal_change : bool = false -> Pedal != pre(Pedal);
assume "no rapid pedal changes" :
  condition pedal_change occurs sporadic with IAT
BSCU.pedal_push_min;
```

Figure 13: Patterns used to constrain the button pushes of the CRP version of the Wheel Braking System.

In contrast, the button press frequency assumptions for the SWPI version of the model are shown in Figure 14. In these assumptions the integers 13 and 20 refer to the number of time steps that occur in reference to the base clock. It is unclear how this number corresponds to the actual timing information of the system (how long are 13 and 20 time steps in real time?). This timing information is made explicit in the assumptions for the CRP version of the model.



```
assume "Button Presses are Long Enough to be Seen by All
Nodes" :
  Agree_Nodes.True_At_Least(Panel.LO,  13) and
  Agree_Nodes.True_At_Least(Panel.MED, 13) and
  Agree_Nodes.True_At_Least(Panel.MAX, 13);

assume "Button Presses are Bounded in Duration" :
  Agree_Nodes.True_At_Most(Panel.LO,  20) and
  Agree_Nodes.True_At_Most(Panel.MED, 20) and
  Agree_Nodes.True_At_Most(Panel.MAX, 20);

assume "Button Presses Do Not Occur Too Quickly" :
  Agree_Nodes.True_At_Least(not Panel.LO,  13) and
  Agree_Nodes.True_At_Least(not Panel.MED, 13) and
  Agree_Nodes.True_At_Least(not Panel.MAX, 13);
```

Figure 14: Constraints used to constrain the button pushes for the SWPI version of the Wheel Braking System.

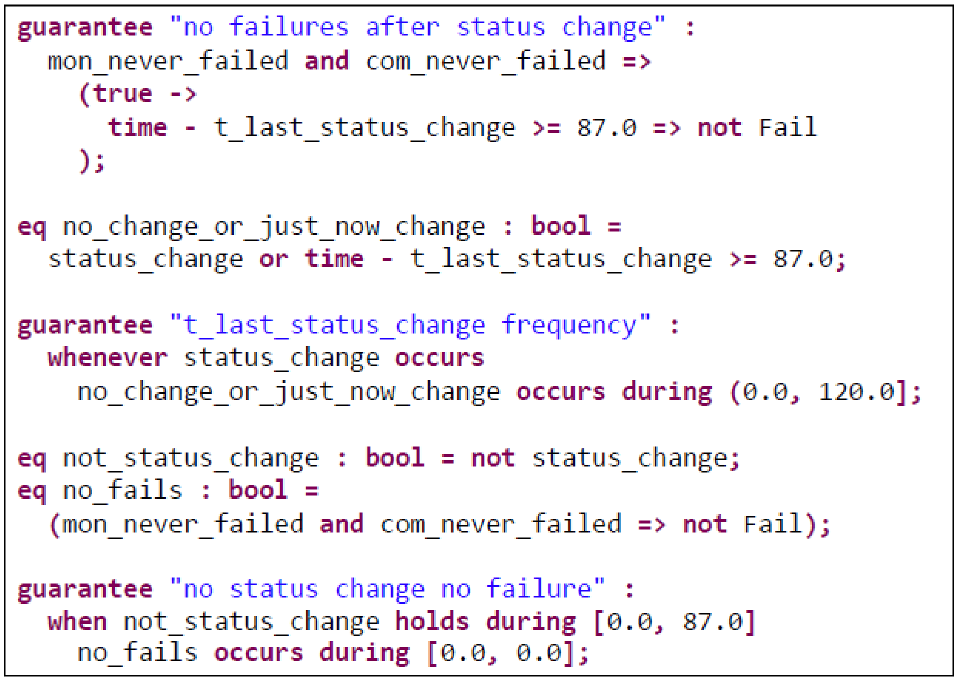
3.2.1.3 Properties of the CRP model

As mentioned earlier, the CRP version of the WBS model proves its properties compositionally. There is an AGREE contract for the BSCU, an AGREE contract for each CHANNEL, an AGREE contract for the MON component and an AGREE contract for the COM component. The contracts of the MON and COM components prove the contract of the CHANNEL components. Likewise, the contracts of the CHANNEL components prove the contracts of the BSCU component.

We prove that neither CHANNEL produces an error if there are no changes on the interfaces (the pedal or the buttons) for more than 87ms. This is also assuming that neither of the MON or COM components has failed. This property is expressed by the guarantees shown in Figure 15.

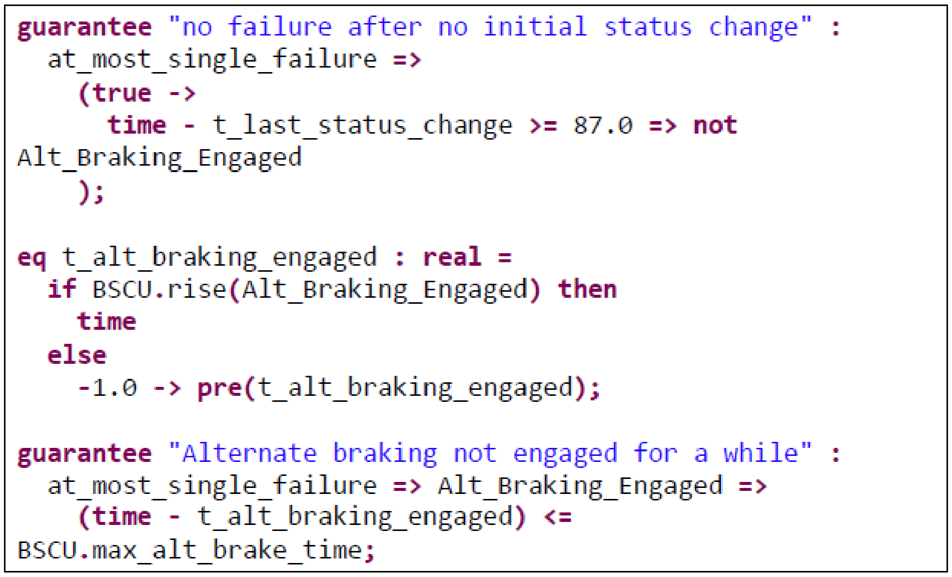
The variable *t_last_status_change* records the time of the last change from a button or the pedal. The first guarantee says if over 87ms have passed since this last change (and if neither the MON nor COM has ever failed) then the channel will not produce a failure. The other two guarantees are used to assert that this event actually triggers with a certain frequency. This is important for proving properties in the parent BSCU component.

The property that we prove for the BSCU is similar to this. We prove that if at most one MON or COM component has failed then at least one channel is error free if the status has not changed for at least 87ms. We also prove that if there has been at most one single failure and if the alternate braking system is engaged then the alternative braking system has only been engaged for a finite amount of time. This indicates that as long as the system has no more than one failure then errors produced by the channel will only be transient. These formalized properties are shown Figure 16.



```
guarantee "no failures after status change" :
  mon_never_failed and com_never_failed =>
    (true ->
      time - t_last_status_change >= 87.0 => not Fail
    );

eq no_change_or_just_now_change : bool =
  status_change or time - t_last_status_change >= 87.0;

guarantee "t_last_status_change frequency" :
  whenever status_change occurs
    no_change_or_just_now_change occurs during (0.0, 120.0];

eq not_status_change : bool = not status_change;
eq no_fails : bool =
  (mon_never_failed and com_never_failed => not Fail);

guarantee "no status change no failure" :
  when not_status_change holds during [0.0, 87.0]
    no_fails occurs during [0.0, 0.0];
```

Figure 15: Properties for each channel of the Wheel Braking System.



```
guarantee "no failure after no initial status change" :
  at_most_single_failure =>
    (true ->
      time - t_last_status_change >= 87.0 => not
Alt_Braking_Engaged
    );

eq t_alt_braking_engaged : real =
  if BSCU.rise(Alt_Braking_Engaged) then
    time
  else
    -1.0 -> pre(t_alt_braking_engaged);

guarantee "Alternate braking not engaged for a while" :
  at_most_single_failure => Alt_Braking_Engaged =>
    (time - t_alt_braking_engaged) <=
BSCU.max_alt_brake_time;
```

Figure 16: The top level properties for the Wheel Braking System.

The analogous requirement for the SWPI version of the model is shown Figure 17.

```
guarantee "At Least One Channel Active - One Failure" :
  At_Most_One_Failed_Component =>
    Agree_Nodes.Duration(not Initializing
      and not(CH1.Active or CH2.Active)) < 7;
```

Figure 17: The SWPI version of the properties from Figure 16.

Once again this requirement is in reference to the time of the base clock of the quasi-synchronous system. It is not clear how the integer 7 corresponds to the real-time constraint of the system. This is made explicit in the properties for the CRP version of the model.

3.2.2 Pilot Flying

A Flight Guidance System (FGS) is a component of the overall Flight Control System (FCS) that compares the measured state of an aircraft (position, speed, and attitude) to the desired state and generates pitch and roll guidance commands to minimize the difference between the measured and desired state. In many aircraft, the Flight Guidance function at the system level is implemented as two physical sides, or channels, one on the left and one on the right side of the aircraft. These redundant implementations communicate with each other over a cross-channel bus as shown in Figure 18.

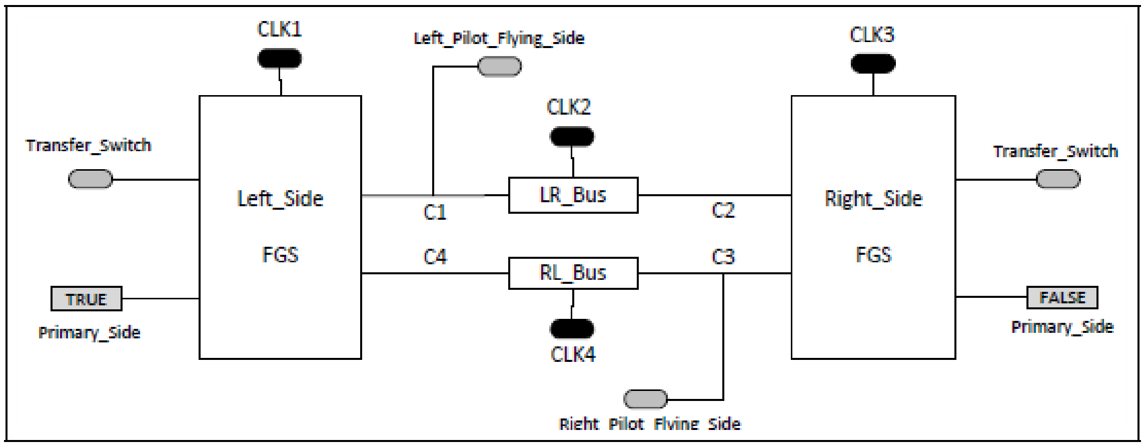


Figure 18: The architecture of a Flight Guidance System.

3.2.2.1 Differences between the SWPI model and CRP model

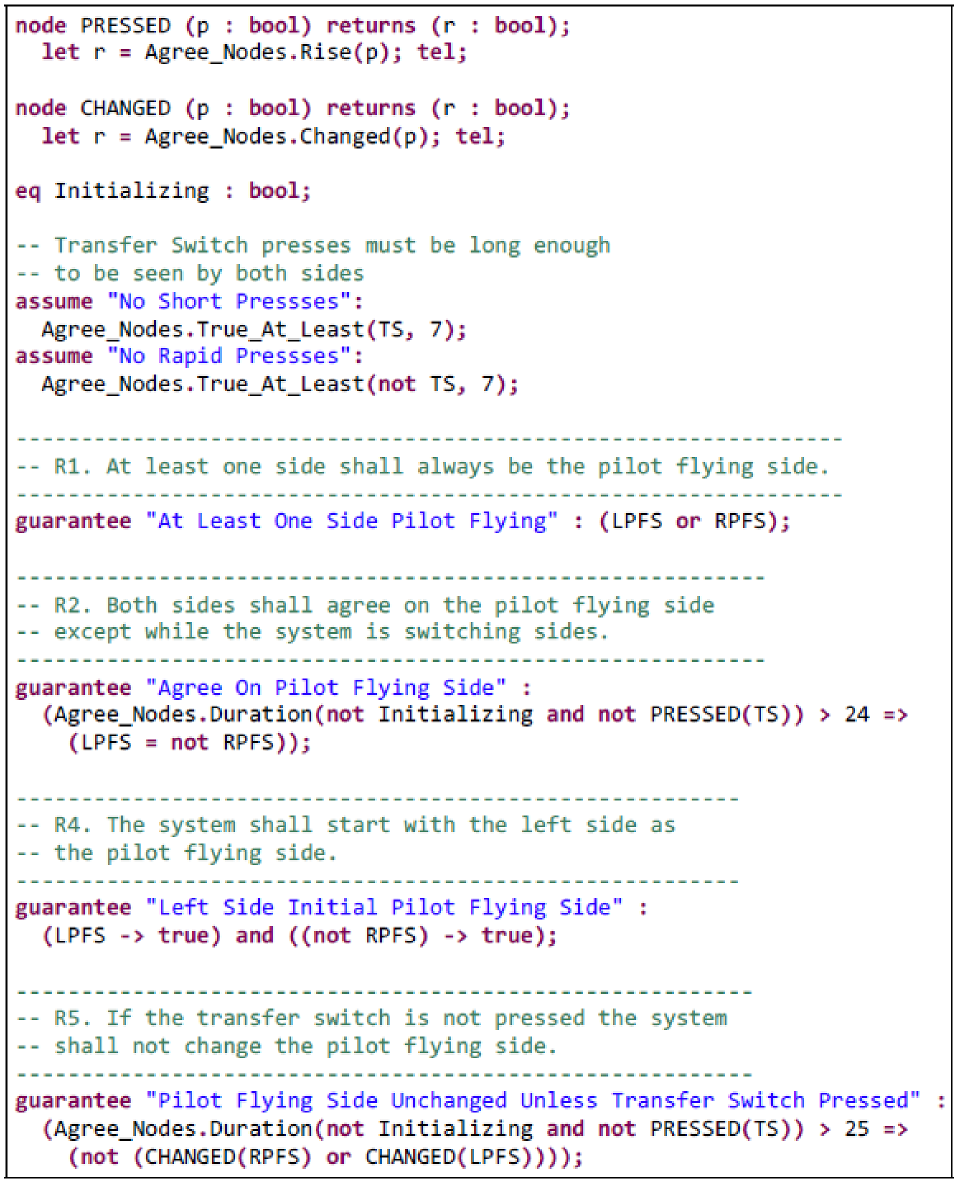
In the final report for the SWIPI project, we discussed different implementations of this system assuming synchronous, asynchronous, and quasi-synchronous architectures. The report discusses results for implementing the quasi-synchronous version. In the CRP project we re-implemented the quasi-synchronous version of the model using real-time patterns. In this version of the architecture each side and each bus runs on its own independent clock. We assume that the buses run at a rate that is faster than the clock of each individual side. However, neither the clocks of the side components nor the bus components are synchronized.

We also simplified the AADL architecture. The SWIPI version of the model contained three components within each side. One component computes the state of the side (either Flying, Inhibited, or Listening). This component is referred to as the "Pilot Flying Side Logic". The remaining two components send a signal to this logic component when the transfer switch state transitions from low to high and when the bus sends a signal indicating that the other side component has transferred from a non-flying state to a flying state.

Rather than explicitly modeling these components that send signals to the logic component when some other external signal transitions from false to true we opted to consolidate the behavior of these three components into a single AGREE contract for the side component. From a verification standpoint this makes the model easier to analyze. We do not believe that this decreases the fidelity between the AADL model and how the model would be physically implemented.

Figure 19 shows the top level contract for the quasi-synchronous version of the model from the SWPI project and Figure 20 shows the top level contract for the CRP version of the model. There are three clear difference between the top level specifications in for each version of the model:

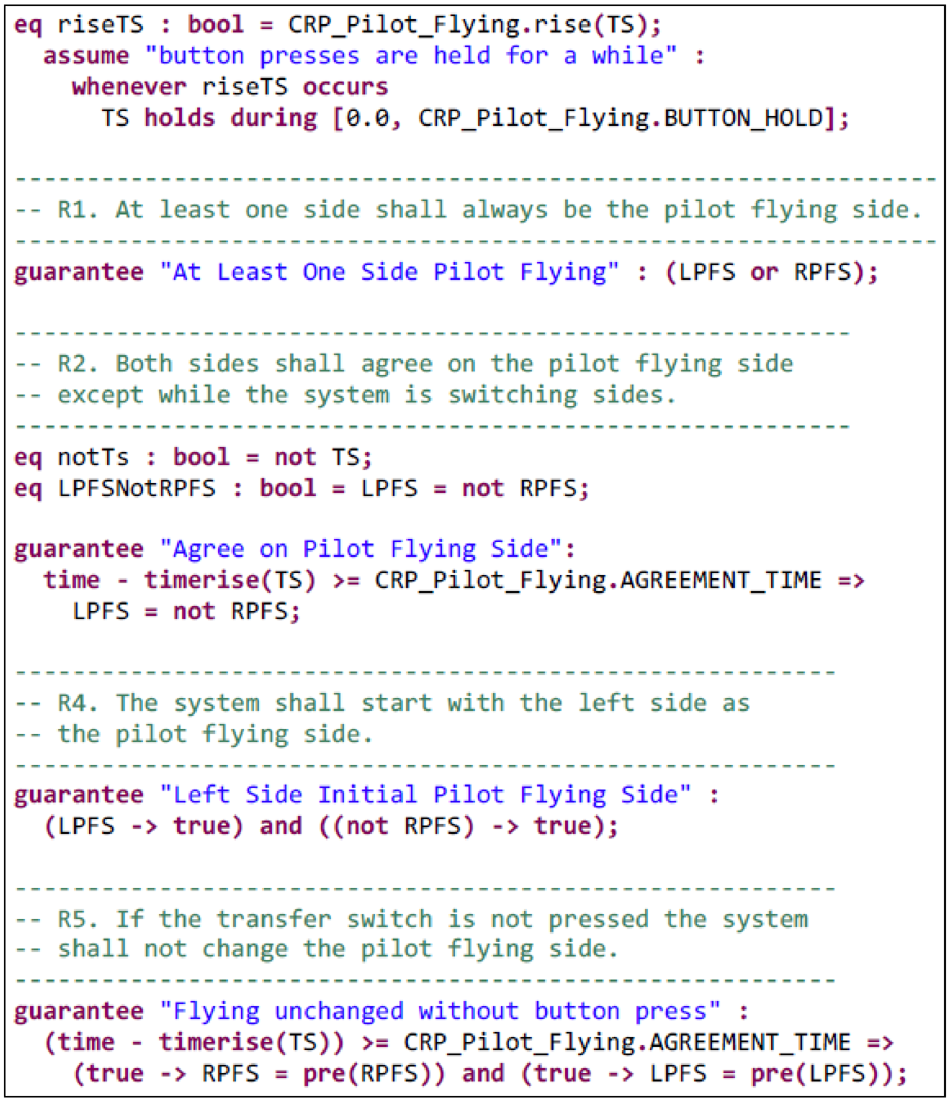
Approved for Public Release; Distribution Unlimited.



```
node PRESSED (p : bool) returns (r : bool);
  let r = Agree_Nodes.Rise(p); tel;

node CHANGED (p : bool) returns (r : bool);
  let r = Agree_Nodes.Changed(p); tel;

eq Initializing : bool;

-- Transfer Switch presses must be long enough
-- to be seen by both sides
assume "No Short Pressses":
  Agree_Nodes.True_At_Least(TS, 7);
assume "No Rapid Pressses":
  Agree_Nodes.True_At_Least(not TS, 7);

----------------------------------------------------------------
-- R1. At least one side shall always be the pilot flying side.
----------------------------------------------------------------
guarantee "At Least One Side Pilot Flying" : (LPFS or RPFS);

----------------------------------------------------------
-- R2. Both sides shall agree on the pilot flying side
-- except while the system is switching sides.
----------------------------------------------------------
guarantee "Agree On Pilot Flying Side" :
  (Agree_Nodes.Duration(not Initializing and not PRESSED(TS)) > 24 =>
    (LPFS = not RPFS));

--------------------------------------------------------
-- R4. The system shall start with the left side as
-- the pilot flying side.
--------------------------------------------------------
guarantee "Left Side Initial Pilot Flying Side" :
  (LPFS -> true) and ((not RPFS) -> true);

---------------------------------------------------------
-- R5. If the transfer switch is not pressed the system
-- shall not change the pilot flying side.
---------------------------------------------------------
guarantee "Pilot Flying Side Unchanged Unless Transfer Switch Pressed" :
  (Agree_Nodes.Duration(not Initializing and not PRESSED(TS)) > 25 =>
    (not (CHANGED(RPFS) or CHANGED(LPFS))));
```

Figure 19: The top level requirements for the SWPI version of the Pilot Flying model.

Approved for Public Release; Distribution Unlimited.



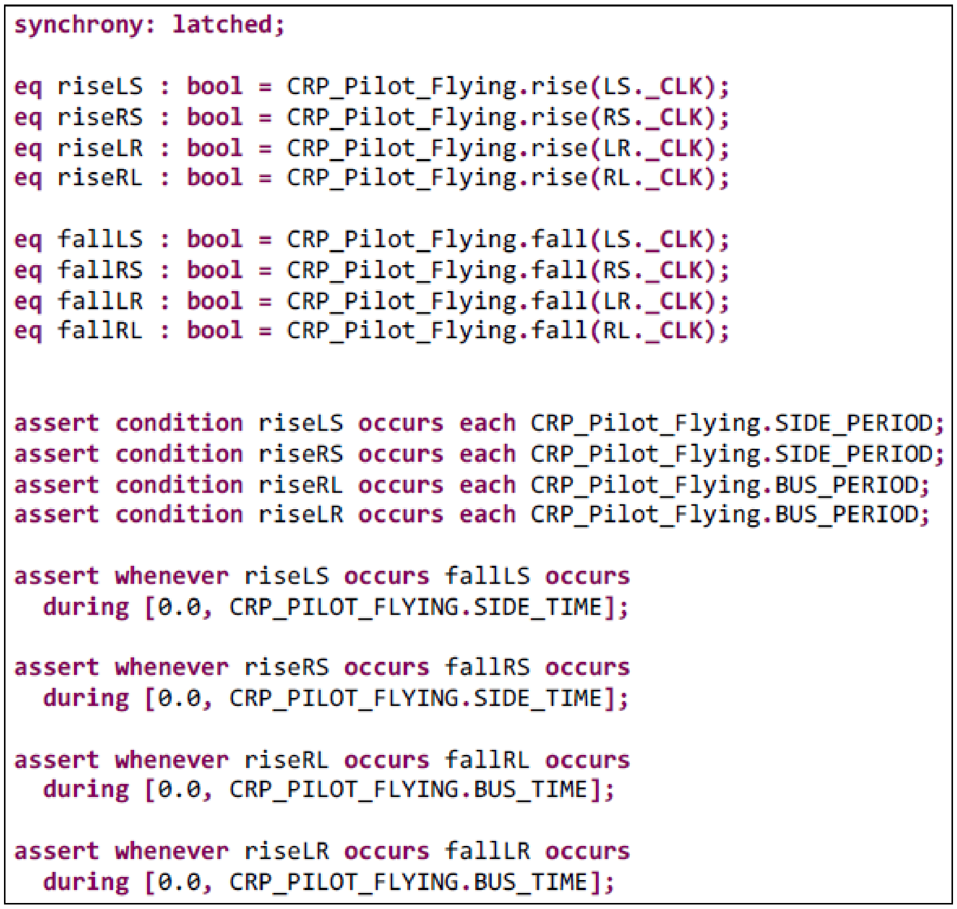
```
eq riseTS : bool = CRP_Pilot_Flying.rise(TS);
  assume "button presses are held for a while" :
    whenever riseTS occurs
      TS holds during [0.0, CRP_Pilot_Flying.BUTTON_HOLD];

---------------------------------------------------------------
-- R1. At least one side shall always be the pilot flying side.
---------------------------------------------------------------
guarantee "At Least One Side Pilot Flying" : (LPFS or RPFS);

---------------------------------------------------------
-- R2. Both sides shall agree on the pilot flying side
-- except while the system is switching sides.
---------------------------------------------------------
eq notTs : bool = not TS;
eq LPFSNotRPFS : bool = LPFS = not RPFS;

guarantee "Agree on Pilot Flying Side":
  time - timerise(TS) >= CRP_Pilot_Flying.AGREEMENT_TIME =>
    LPFS = not RPFS;

--------------------------------------------------------
-- R4. The system shall start with the left side as
-- the pilot flying side.
--------------------------------------------------------
guarantee "Left Side Initial Pilot Flying Side" :
  (LPFS -> true) and ((not RPFS) -> true);

--------------------------------------------------------
-- R5. If the transfer switch is not pressed the system
-- shall not change the pilot flying side.
--------------------------------------------------------
guarantee "Flying unchanged without button press" :
  (time - timerise(TS)) >= CRP_Pilot_Flying.AGREEMENT_TIME =>
    (true -> RPFS = pre(RPFS)) and (true -> LPFS = pre(LPFS));
```

Figure 20: The top level requirements for the CRP version of the Pilot Flying model.

1. The SWPI version of the model references integer bounds referring to the number of clock steps in the model. In contrast the CRP version of the model contains bounds in reference to real-time. This makes the specifications in the CRP version of the model easier to understand.
2. The specifications in the CRP version of the model do not distinguish whether or not the system is “initializing”. During the SWIPI project we did not have a way to constrain the initial output values of components before their clocks have ticked. We have since added new constructs to the language so we no longer need to tick each component’s clock to initialize their outputs.
3. The CRP version of the model has weaker assumptions about how frequently the transfer switch is pressed. In the CRP version of the model the button must be held down (set to

true) for at least 100ms, but there are no constraints on how long the button needs to be depressed.

The relationships between the clocks are represented very concisely using assertions and patterns in the top level AADL implementation of the model. These constraints are shown in Figure 21.



```
synchrony: latched;

eq riseLS : bool = CRP_Pilot_Flying.rise(LS._CLK);
eq riseRS : bool = CRP_Pilot_Flying.rise(RS._CLK);
eq riseLR : bool = CRP_Pilot_Flying.rise(LR._CLK);
eq riseRL : bool = CRP_Pilot_Flying.rise(RL._CLK);

eq fallLS : bool = CRP_Pilot_Flying.fall(LS._CLK);
eq fallRS : bool = CRP_Pilot_Flying.fall(RS._CLK);
eq fallLR : bool = CRP_Pilot_Flying.fall(LR._CLK);
eq fallRL : bool = CRP_Pilot_Flying.fall(RL._CLK);


assert condition riseLS occurs each CRP_Pilot_Flying.SIDE_PERIOD;
assert condition riseRS occurs each CRP_Pilot_Flying.SIDE_PERIOD;
assert condition riseRL occurs each CRP_Pilot_Flying.BUS_PERIOD;
assert condition riseLR occurs each CRP_Pilot_Flying.BUS_PERIOD;

assert whenever riseLS occurs fallLS occurs
  during [0.0, CRP_PILOT_FLYING.SIDE_TIME];

assert whenever riseRS occurs fallRS occurs
  during [0.0, CRP_PILOT_FLYING.SIDE_TIME];

assert whenever riseRL occurs fallRL occurs
  during [0.0, CRP_PILOT_FLYING.BUS_TIME];

assert whenever riseLR occurs fallLR occurs
  during [0.0, CRP_PILOT_FLYING.BUS_TIME];
```

Figure 21: The assertions constraining the behavior of the clocks in the CRP version of the Pilot Flying Model.

Similar to the CRP version of the WBS model we picked somewhat arbitrary parameters for the periods of the side components and bus components. However, we made sure that the bus component would execute at least twice since each side component. We chose the side periods to be 20ms, the bus periods to be 10ms, the maximum side execution time to be 5ms, and the maximum bus execution time to be 2ms. Based on these timing values we inferred that the transfer switch would likely need to be held down for at least 100ms before both sides could reach agreement on who was the pilot flying side.

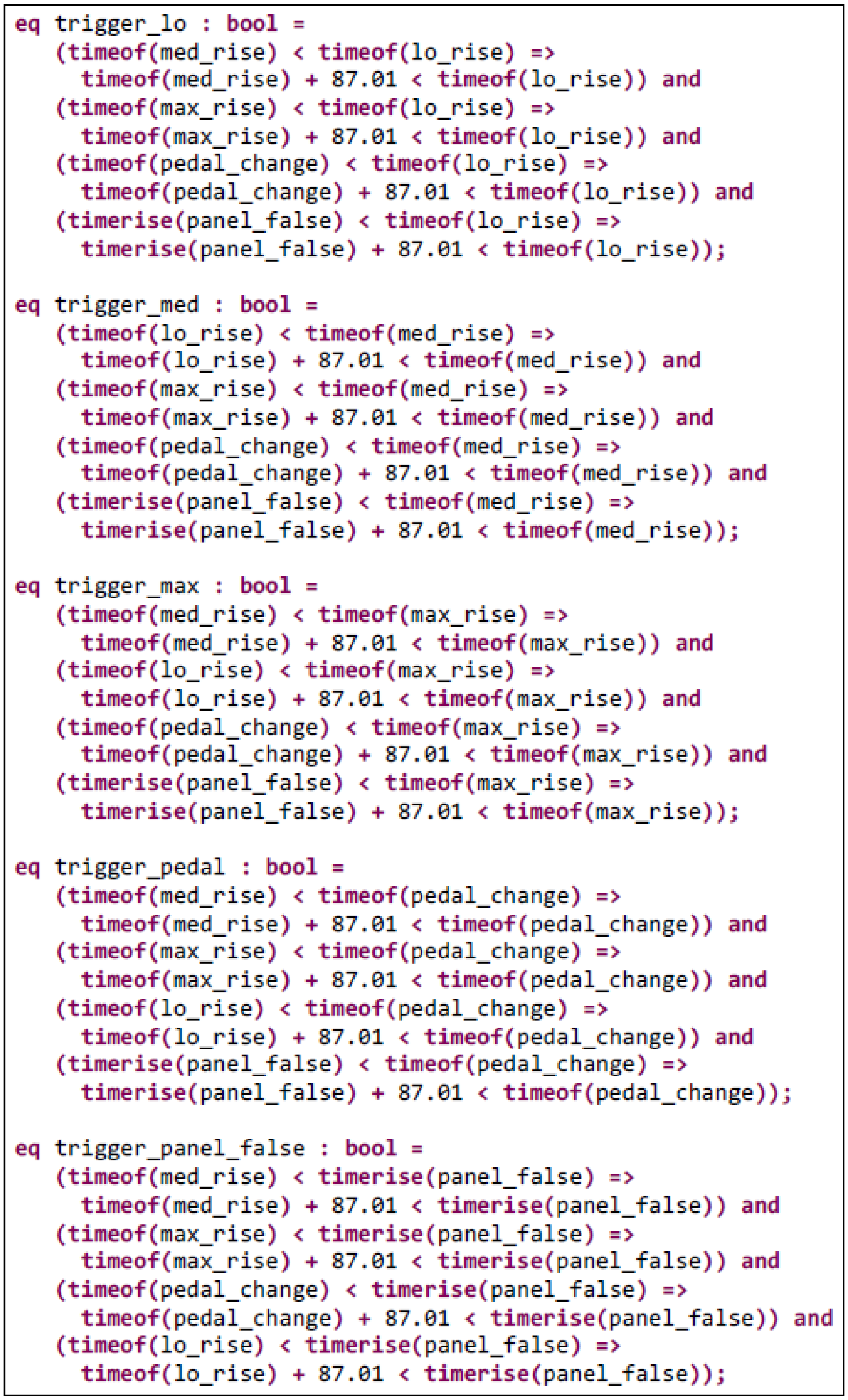
4. Results and Discussion

4.1 Wheel Braking System

The analysis time of the properties for the CRP version of the Wheel Braking System has improved significantly. Using a laptop computer with an Intel CoreTM i7 running at 2.7 GHz it takes roughly 8 minutes to prove the top level guarantees of the quasi-synchronous version of the model. In contrast, on the same machine it takes less than 2 minutes to prove the top level requirements of the CRP version of the model and less than two minutes to prove the requirements of one of the channel components. The quasi-synchronous version of the model takes longer to analyze because of the exponential increase in the possible number of interleaving's between different component clocks. In the quasi-synchronous version of the model each component's clock can tick at most twice since any other component in the system. Loosening this constraint dramatically increases the runtime of the model.

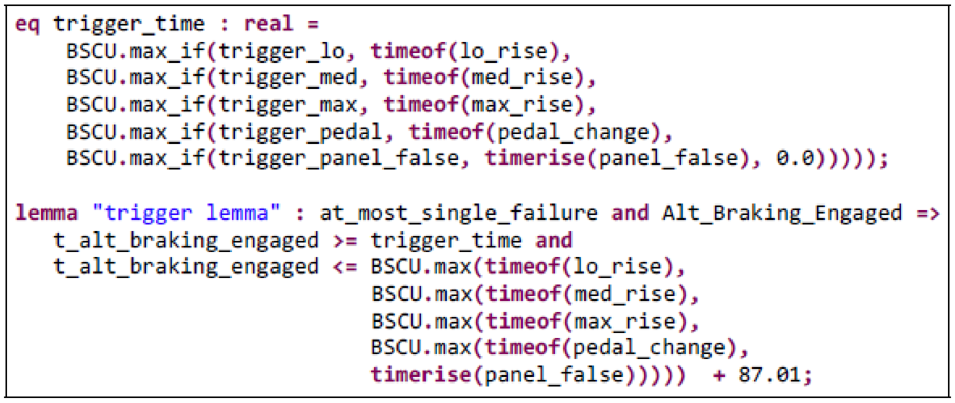
The CRP version of the model does not suffer the same performance consequences. However, a significant amount of effort was taken in order to figure out the correct lemmas needed in the model to prove the top level requirements. This task was not only time consuming, it was intellectually challenging to complete. We discovered these lemmas by inspecting the counterexamples of the inductive check from the model checker to determine what additional facts the system needed to learn in order to eliminate the counterexamples. The constraints for the lemma that was most challenging to discover is shown in Figures 22 and 23. These constraints are used to define a "trigger time" which determines the time range in which the alternate braking system would have engaged in response to a particular input (either a button press or change in pedal pressure). It took significant insight into the behavior of the system in order to determine these constraints.

Approved for Public Release; Distribution Unlimited.



```
eq trigger_lo : bool =
   (timeof(med_rise) < timeof(lo_rise) =>
     timeof(med_rise) + 87.01 < timeof(lo_rise)) and
   (timeof(max_rise) < timeof(lo_rise) =>
     timeof(max_rise) + 87.01 < timeof(lo_rise)) and
   (timeof(pedal_change) < timeof(lo_rise) =>
     timeof(pedal_change) + 87.01 < timeof(lo_rise)) and
   (timerise(panel_false) < timeof(lo_rise) =>
     timerise(panel_false) + 87.01 < timeof(lo_rise));

eq trigger_med : bool =
   (timeof(lo_rise) < timeof(med_rise) =>
     timeof(lo_rise) + 87.01 < timeof(med_rise)) and
   (timeof(max_rise) < timeof(med_rise) =>
     timeof(max_rise) + 87.01 < timeof(med_rise)) and
   (timeof(pedal_change) < timeof(med_rise) =>
     timeof(pedal_change) + 87.01 < timeof(med_rise)) and
   (timerise(panel_false) < timeof(med_rise) =>
     timerise(panel_false) + 87.01 < timeof(med_rise));

eq trigger_max : bool =
   (timeof(med_rise) < timeof(max_rise) =>
     timeof(med_rise) + 87.01 < timeof(max_rise)) and
   (timeof(lo_rise) < timeof(max_rise) =>
     timeof(lo_rise) + 87.01 < timeof(max_rise)) and
   (timeof(pedal_change) < timeof(max_rise) =>
     timeof(pedal_change) + 87.01 < timeof(max_rise)) and
   (timerise(panel_false) < timeof(max_rise) =>
     timerise(panel_false) + 87.01 < timeof(max_rise));

eq trigger_pedal : bool =
   (timeof(med_rise) < timeof(pedal_change) =>
     timeof(med_rise) + 87.01 < timeof(pedal_change)) and
   (timeof(max_rise) < timeof(pedal_change) =>
     timeof(max_rise) + 87.01 < timeof(pedal_change)) and
   (timeof(lo_rise) < timeof(pedal_change) =>
     timeof(lo_rise) + 87.01 < timeof(pedal_change)) and
   (timerise(panel_false) < timeof(pedal_change) =>
     timerise(panel_false) + 87.01 < timeof(pedal_change));

eq trigger_panel_false : bool =
   (timeof(med_rise) < timerise(panel_false) =>
     timeof(med_rise) + 87.01 < timerise(panel_false)) and
   (timeof(max_rise) < timerise(panel_false) =>
     timeof(max_rise) + 87.01 < timerise(panel_false)) and
   (timeof(pedal_change) < timerise(panel_false) =>
     timeof(pedal_change) + 87.01 < timerise(panel_false)) and
   (timeof(lo_rise) < timerise(panel_false) =>
     timeof(lo_rise) + 87.01 < timerise(panel_false));
```

Figure 22: The "triggering" constraints needed to prove the top level properties of the CRP version of the WBS model.



```
eq trigger_time : real =
    BSCU.max_if(trigger_lo, timeof(lo_rise),
    BSCU.max_if(trigger_med, timeof(med_rise),
    BSCU.max_if(trigger_max, timeof(max_rise),
    BSCU.max_if(trigger_pedal, timeof(pedal_change),
    BSCU.max_if(trigger_panel_false, timerise(panel_false), 0.0)))));

lemma "trigger lemma" : at_most_single_failure and Alt_Braking_Engaged =>
   t_alt_braking_engaged >= trigger_time and
   t_alt_braking_engaged <= BSCU.max(timeof(lo_rise),
                            BSCU.max(timeof(med_rise),
                            BSCU.max(timeof(max_rise),
                            BSCU.max(timeof(pedal_change),
                            timerise(panel_false)))))  + 87.01;
```

Figure 23: The "triggering" lemma needed to prove the top level properties of the CRP version of the WBS model.

4.2 Pilot Flying

While we were able to report a faster analysis time for the CRP version of the WBS model, the analysis time for the pilot flying model is much slower. This is due to two reasons:

1. The architecture that we chose for the model is not compositional. We could possibly improve this by placing each side and bus pair into another component. However we would need to develop a contract for this new component.
2. We have not yet discovered the lemmas that we need to prove all of the properties of the model.

Using a laptop computer with an Intel CoreTMi7 running at 2.7 GHz it takes roughly 6 minutes to prove all of the properties of the SWPI version of the model. However, on the same machine we are only able to prove Requirements 1 and 4 of the CRP version of the model. Requirement 4 proves in less than a minute, but Requirement 1 takes over an hour. At this time we are unable to produce proofs for the remaining requirements for the model. With more time spent on discovering lemmas we could likely achieve similar runtimes for this model as we did with the CRP version of the WBS model.

5. Conclusions

The main goal of this project was to investigate the use of patterns to specify formal requirements for timed systems. We found that many of the specifications that we developed for the examples from the SWPI project where unintuitive or hard to understand. We argue that these unintuitive specifications are a bi-product of modeling systems quasi-synchronously. This is because the notion of time for these systems is very abstract; we do not model how quickly events occur in terms of real-time. Timing of events is constrained in reference to how often one clock in the system ticks with respect to another. Therefore we cannot specify that a button must be held for a specific number of seconds. Instead how long a button must be pressed needs to be in reference to some base rate. This can vastly over-approximate the necessary bounds needed for the property to hold in a real system.

For example, consider a system of 10 components running on separate clock domains that all receive the same input. Assume that each clock runs at 10ms and has 1ms of jitter with no drift. If one were to model this system using quasi-synchronous constraints they would assume each clock ticks no more than once since any other clock ticks. In order for all components to receive the same input the value must remain constant for at least 10 ticks. If one were to translate this constraint from quasi-synchronous ticks into real-time they would conclude that inputs must remain constant for at-least 10*11ms = 110ms. However, without making the quasi-synchronous abstraction it should be clear that all components should see the same input as long as it is held for at least 11ms (the period of each clock plus the jitter).

In order to make the specification language easier to understand it was clear that properties should be specified in terms of real-time constraints. This example also demonstrates the loss of fidelity that one gets when making quasi-synchronous abstractions. This concern led us to implement the real-time patterns from the CESAR project [17] into the AGREE tool. However, through the course of the project we have discovered that increased modeling fidelity and clarity of specifications has come with the tradeoff of more difficult proofs. Much of the time spent developing the examples for this project was used on discovering the correct lemmas needed to prove the properties of interest. Once the lemmas were discovered verification became more tractable on the CRP versions of the models than the SWPI versions.

For future work we plan to improve and automate the task of real-time lemma discovery. This could dramatically decrease the amount of time needed to prove specifications. This would also make AGREE easier to use for users who do not have formal methods experience.

References

1. CESAR: Definition and exemplification of RSL and RMM. Technical report, Cost-efficient methods and processes for safety relevant embedded systems (2010)

http://www.cesarproject.eu/fileadmin/user upload/CESAR D SP2 R2.1 M1 v1.000.pdf.

2. Dwyer, M.B., Avrunin, G.S., Corbett, J.C.: Patterns in property specifications for finite-state verification. In: ICSE, IEEE (1999) 411–420

3. Konrad, S., Cheng, B.H.: Real-time specification patterns. In: Proceedings of the 27th international conference on Software engineering, ACM (2005) 372–381

4. Gruhn, V., Laue, R.: Patterns for timed property specifications. Electronic Notes in Theoretical Computer Science 153 (2006) 117–133

5. Bellini, P., Nesi, P., Rogai, D.: Expressing and organizing real-time specification patterns via temporal logics. Journal of Systems and Software 82 (2009) 183–196

6. Reinkemeier, P., Stierand, I., Rehkop, P., Henkler, S.: A pattern-based requirement specification language: Mapping automotive specific timing requirements. In: Fachtagung des GI-Fachbereichs Softwaretechnik. (2011) 99–108

7. Etzien, C., Gezgin, T., Froschle, S., Henkler, S., Rettberg, A.: Contracts for evolving systems. In: ISORC. (2013) 1–8

8. Alur, R.: Techniques for automatic verification of real-time systems. PhD thesis, Stanford University (1991)

9. Koymans, R.: Specifying real-time properties with metric temporal logic. Real-time systems 2 (1990) 255–299

10. Moser, L.E., Ramakrishna, Y., Kutty, G., Melliar-Smith, P.M., Dillon, L.K.: A graphical environment for the design of concurrent real-time systems. ACM Trans- actions on Software Engineering and Methodology (TOSEM) 6 (1997) 31–79

11. Abid, N., Dal Zilio, S., Le Botlan, D.: Real-time specification patterns and tools. In: Formal Methods for Industrial Critical Systems. Springer (2012) 1–15

12. Cofer, D.D., Gacek, A., Miller, S.P., Whalen, M.W., LaValley, B., Sha, L.: Com- positional verification of architectural models. In Goodloe, A.E., Person, S., eds.: NFM. Volume 7226, Berlin, Heidelberg, Springer-Verlag (2012) 126–140

13. Feiler, P.H., Gluch, D.P.: Model-Based Engineering with AADL: An Introduction to the SAE Architecture Analysis & Design Language. 1st edn. Addison-Wesley Professional (2012)

14. Halbwachs, N., Caspi, P., Raymond, P., Pilaud, D.: The synchronous dataflow programming language LUSTRE. In: Proceedings of the IEEE. (1991) 1305–1320

15. Backes, J.D., Cofer, D., Miller, S., Whalen, M.: Requirements analysis of a quad-redundant flight control system. In: NFM. (2015) 82–96

Approved for Public Release; Distribution Unlimited.

16. Murugesan, A., Heimdahl, M.P., Whalen, M.W., Rayadurgam, S., Komp, J., Duan, L., Kim, B.G., Sokolsky, O., Lee, I.: From requirements to code: Model based development of a medical cyber physical system. SEHC (2014)

17. CESAR: The CESAR project. http://www.cesarproject.eu/ (2010)

18. Dutertre, B., Sorea, M.: Timed systems in SAL. Technical report, SRI International (2004)

19. Pike, L.: Real-time system verification by k-induction. Technical report, NASA (2005)

20. Gao, J., Whalen, M., Van Wyk, E.: Extending lustre with timeout automata. In: SLA++P. (2007)

21. Gacek, A., Backes, J., Whalen, M.W., Cofer, D.: AGREE Users Guide[5] (2014)

22. G´omez, R., Bowman, H.: Efficient Detection of Zeno Runs in Timed Automata. In: FORMATS. Springer Berlin Heidelberg (2007) 195–210

23. JKind: A Java implementation of the KIND model checker[6]. (2013)

24. Gafni, V., Benveniste, A., Caillaud, B., Graf, S., Josko, B.: Contract specification language (CSL). Technical report, SPEEDS Deliverable D.2.5.4 (2008)

25. Pike, L.: Modeling time-triggered protocols and verifying their real-time schedules. In: Formal Methods in Computer-Aided Design. (2007) 231–238

26. Sorea, M., Dutertre, B., Steiner, W.: Modeling and verification of time-triggered communication protocols. In: Object Oriented Real-Time Distributed Computing (ISORC), 2008 11th IEEE International Symposium on, IEEE (2008) 422–428

27. Li, W., Grard, L., Shankar, N.: Design and verification of multi-rate distributed systems. In: Formal Methods and Models for Codesign (MEMOCODE), 2015 ACM/IEEE International Conference on. (2015) 20–29

[5] Available at: http://github.com/smaccm/smaccm
[6] Available at: http://github.com/agacek/jkind

List of Symbols, Abbreviations and Acronyms

AADL	Architectural Analysis and Design Language
AGREE	Assume Guarantee Reasoning Environment
BSCU	Braking System Control Unit
CESAR	Cost-efficient Methods and Processes for Safety-relevant Embedded Systems
COM	Command
CRP	Contract Requirements Patterns
CSL	Contract Specification Language
FCS	Flight Control System
FGS	Flight Guidance System
LTL	Linear Temporal Logic
MON	Monitor
MTL	Metric Temporal Logic
OSATE	Open Source AADL Tool Environment
QS	Quasi-synchronous
RMM	Requirement Meta-Model
RSL	Requirements Specification Language
RTGIL	Real-Time Graphical Interval Logic
SMT	Satisfiability Modulo Theories
SWPI	Software Productivity Initiative
TCTL	Timed Computational Tree Logic
TTS	Timed Transition System
TILCOX	Extended Temporal Interval Logic
WBS	Wheel Braking System

Approved for Public Release; Distribution Unlimited.